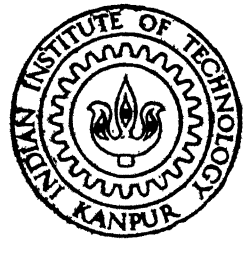
Design And Synthesis of Asynchronous Digital Circuits

by
AJAY KUMAR GARG



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
JANUARY, 1998

Design And Synthesis of Asynchronous Digital Circuits

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Ajay Kumar Garg

to the

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

Jan 1998

EE-1998-M-GAR-DES

Entered in System

Am
7-4-98

CERTIFICATE

This is to certify that the work contained in the thesis entitled "Design and Synthesis of Asynchronous Digital Circuits", by **Ajay Kumar Garg** has been carried out under my supervision and this work has not been submitted elsewhere for a degree.

19TH JANUARY 1998

Dr Subir K. Roy
Department of Electrical Engineering
Indian Institute of Technology, Kanpur

Acknowledgement

I am indebted to my thesis adviser, Dr. Subir K. Roy, for introducing me to this fascinating world of asynchronous logic design and synthesis. I express my deepest gratitude to him for providing me all the support, patiently listening to my problems and providing valuable suggestions.

I am also grateful to ARCUS Technology Limited for providing BSIM II model parameters to carry out the various simulations. I am also very thankful to Mr Ashish of Silicon Automation System Bangalore for providing FPGA synthesis tool. I am also thankful to Mr S.K.Desai for his assistance. My special thanks go to Dr O.P. Nijhawan and Dr A.K. Jaiswal, I.R.D.E. Dehradun, for giving me the opportunity to pursue higher studies.

I am also thankful to all my friends who made my stay memorable at I.I.T. Kanpur and my special thanks to Bupesh, Pradip and Biplab for their constant encouragement and support. Finally I want to thank my parents for their moral support and encouragement.

Abstract

Asynchronous design has been an active area of research since early 60's. There has been a resurgence of interest in the design of asynchronous circuits in recent years, due to their ability to eliminate clock skew problems, achieve average case performance, automatic adapt automatically to processing and environmental variations and provide a better technology migration route as compared to synchronous circuits.

There are several implementation approaches for asynchronous digital circuits based upon the delay model used. These models are the bounded delay model, delay insensitive model, speed independent model and the quasi delay insensitive model. In the present thesis, we describe the methodology for the designing asynchronous digital systems based on a delay insensitive model using the Non Return to Zero (NRZ) event driven signaling protocol. We use this approach to design an asynchronous arbiter and the bus based interconnect structure for bundled data transfer. Desai [23] has given the different implementations of the bus based interconnect structures. But the implementation presented here has the advantage of reducing the bus width and the decoder complexity.

Several methodology exists for synthesizing delay insensitive digital circuits e.g., trace theory and transforming asynchronous circuits as communicating processes (CP) to delay insensitive circuit using CSP language. We synthesize delay insensitive circuits from its behavioral description written in a hardware description language (HDL) such as Verilog and VHDL. The design is represented in a HDL and its control and data flow graph (CDFG) is generated. We describe two algorithmss ***Event List Scheduling*** (ELS) and the ***Modified Event List Scheduling*** (MELS) for scheduling. allocation and binding of resources in an asynchronous digital systems. We test these algorithms on several high level synthesis benchmark examples. Finally we synthesize the various asynchronous circuit modules for a Xilinx FPGA target architecture.

Contents

List Of Figures

List of Table

Chapter 1

Introduction

Most of the today's digital design is based upon the assumption that all the signals are binary and that time is discrete. Both of these assumptions are made in order to simplify the logic design. By assuming binary values on signals, simple Boolean logic can be used to describe and manipulate logic constructs. By assuming time is discrete, hazards and feedback can largely be ignored. However, any system which can operate without these assumptions has the potential to generate better and faster results.

Asynchronous digital circuits keeps the assumption that signals are binary but remove the restriction on time. Hence asynchronous circuits are defined as those in which synchronization is performed without a global signal (usually referred to as the clock). Recently there has been a resurgence of interest in the design of asynchronous circuits due to several advantages they offer over their synchronous counterparts. These are :

- ***Elimination of clock skew problem*** : As system becomes larger, due to higher degree of integration, an increasingly amount of design effort becomes necessary to guarantee minimal skew in the arrival time of the global clock signal at different parts of the chip. In the DEC Alpha microprocessor, nearly a third of silicon area is required for the clock distribution network [2]. This problem worsens as the clock speed increases. In asynchronous circuit skew in synchronization signals can be tolerated, so this extra circuitry is not necessary.

• *Average case performance*: In synchronous system, the performance is dictated by the worst case condition. The clock period must be set long enough to accommodate the slowest operation even though the average delay of the operation is often shorter. In asynchronous circuits speed of circuit is allowed to change dynamically, so the performance is governed more by the average delay rather than the worst case delay, enabling faster operations than the synchronous implementations.

• *Adaptivity to processing and environmental variations*: The delay of VLSI circuit can vary significantly over different fabrication process runs, supply voltages and operating temperatures. For this reason synchronous designs are simulated over a wide variation of these parameters, and clock is set so that majority of chips produced operate correctly under some allowable variations in the above parameters. Due to their adaptive nature asynchronous circuits operate under all variations and simply speeds up or slows down as dictated by these parameters.

• *Component modularity*: In an asynchronous system, components can be interfaced and interconnected without the difficulties associated with a global synchronizing clock in synchronous systems. Also, when a new and faster component becomes available because of the improvement in technology, it can easily replace the slower component in the existing system without requiring any changes in the rest of the system to get an improvement in the system performance.

• *Lower system power requirements*: Asynchronous circuits reduce synchronization power by not requiring additional clock drivers and buffers to limit clock skew. They can also automatically power down unused components. In many synchronous applications, more than half of the power is wasted due to spurious transitions. Asynchronous circuits when designed properly have no spurious transitions. For example, even though a floating point unit on a processor might not be used in a given instruction stream, the unit still must be operated by the clock in a synchronous system. However, in an asynchronous system, a unit is active only when it receives a transition on its input lines and, otherwise, it draws very less power. So asynchronous circuits can make use of a power supply more efficiently.

While asynchronous design has long been used in interface circuits, they are now being considered for the design of high performance processors [4,5,6] and low power

embedded controllers [7]. However, the advantages of asynchronous circuits have not been fully utilized for various reasons.

• ***Lack of mature CAD tools***: Though there has been a rapid development in the VLSI design tools in the recent years. However, these CAD tools, are limited to implementing synchronous designs. While many asynchronous design methods have supporting CAD tools but these are largely in the experimental stage.

• ***Large area overhead for the removal of hazards***: A hazard is a spurious signal transition, or glitch. While hazards can be ignored in a synchronous design as they are filtered out by the clock signal, any hazard in an asynchronous design can potentially lead to malfunctioning. Therefore, careful design is necessary to avoid hazards in an asynchronous design which often leads to a significant increase in Silicon area required to implement the circuit.

• ***Difficulty in interfacing with existing synchronous design***: Many asynchronous design methods require the ability to slow down the environment by withholding an acknowledgment. When interfacing with a synchronous design, this is typically not possible.

There are several disadvantages associated with the design of asynchronous digital circuits. Some of these are;

• ***Necessity for custom design***: As the pace of technology increases, the life spans of products decrease, forcing the VLSI designers to often turn to semi-custom implementation methods based on standard cells and gate arrays, to reduce the time to market. However, many asynchronous design procedures require the use of special complex gates and hence rapid prototyping sometimes becomes difficult.

• ***Unreliable design***: In order to get efficient implementations, many assumptions are made in the design the asynchronous circuits. These implementations must be checked thoroughly by simulating them. Unfortunately, simulation being an imperfect approach, unreliable designs can still be produced.

Several methodologies exist for the design of asynchronous digital systems [1]. These methodologies are characterized by their timing model. The major timing models are defined as follows.

• ***Bounded delay Model:*** Using this model, circuits are designed in the same way as synchronous circuits. It assumes that environment must wait long enough for the circuit to stabilize before inputs are changed [12]. To achieve this it is assumed that the gate and the wire delays are bounded. By assuming that circuits has a bounded delay, it is now possible to construct asynchronous circuits which interface with synchronous environment such as DRAM controller. Since the fundamental mode assumption must be guaranteed to hold under all operating conditions, these circuits may not take the full advantage of variations in delay such as those resulting from data dependencies. Also these methods limit the concurrency within a circuit. None of the methodologies available for this delay model address system design issues. Hence synthesis methods based on this model have not evolved.

• ***Delay Insensitive Model:*** The delay insensitive model assumes that delays in both circuit elements and the wire are finite but unbounded and the correctness of the circuit is independent of both data and wire delays. The major advantage of delay insensitive design is its modularity. Delay insensitive modules are easily composed without the need to worry about gate or wire delays. They are very robust and can achieve an average case performance. However, there are serious disadvantages as well. There can be a large area and the delay overheads to achieve delay insensitivity. A method for delay insensitive circuit design has been proposed by Ebergen [3], which is based on the trace theory. Another method of transforming asynchronous systems specified as communicating process to delay insensitive circuits using the CSP language has been proposed by A.J.Martin [4,32].

• ***Quasi Delay Insensitive Model & Speed independent model***: Many methodologies have been proposed for the synthesis of quasi delay insensitive and speed independent circuits in which the correctness is independent of gate delays but delays of certain wire called isochronic forks are negligible. The main difference between a quasi delay insensitive and a speed independent circuit is that all forks must be isochronic in speed independent circuits. The primary advantage of the quasi delay insensitive and the speed independent circuits is the ability to map the design to basic combinational logic gates such as AND, OR, C element, etc., thus allowing semi custom implementations. However, careful design is necessary to guarantee that the isochronic fork assumption is met. The delay assumptions may not be valid for all technologies e.g field programmable

gate arrays (FPGA), where the wire delay usually dominates. Also they can not be interfaced with the synchronous environments.

One of the major motivation of our work is to synthesize asynchronous digital systems based upon the delay insensitive model, from their behavioral description written in a hardware description language such as Verilog or VHDL. Further we assume that all the circuits are designed based on the two phase, non return to zero (NRZ), event driven methodology and a request acknowledgment protocol is maintained for intermodule communication [22,23]. It is assumed that both data and control signals are transitions. We treat both $0 \rightarrow 1$ and $1 \rightarrow 0$ as identical transitions. Data signals are two rails (r_1, r_0). Data value `1' indicates a transition on rail $r_1$ and data value `0' indicates a transition on rail r_0. Control signals are single rail, representing a transition on their respective control wire. Non return to zero transition signaling gives better performance compared to the return to zero (RZ) transition signaling due to the presence of idle phase in the RZ transition signaling [6].

Design and implementation of the asynchronous system design based on the delay insensitive methodology necessitates the design of a library comprising of the basic modules designed using this protocol. Some of these modules such as U-Gate, U_Reg, Event counter, Toggle flip flop etc. have been designed using this methodology [21, 22, 23]. These basic modules have been employed to design various asynchronous digital system based upon the delay insensitive methodology.

In chapter 2 of this thesis, we describe one such asynchronous design (asynchronous arbiter) based upon this approach. Also bundled data transfer based upon the Sutherland micropiplines approach [8], has been employed for architectures using bus structure for interconnections. Chapter 3 of the thesis describes the high level synthesis approach of asynchronous digital systems from their behavioral description written in a Hardware Description Language (HDL). Here we specify behaviorally the given system in a HDL and synthesize them from its control and data flow graph (CDFG). Two algorithms, event list scheduling [14], and our modification of it have been discussed and implemented for asynchronous scheduling, allocation and binding of the resources. In chapter 4, implementation of various modules discussed above have been synthesized using an RTL synthesis tool and their Xilinx netlist format (XNF) output has been generated for Xilinx XC4000 [29] series of FPGA target architecture.

Chapter 2

Asynchronous Systems

This chapter describes the methodology for the designing asynchronous digital systems based on a event driven delay insensitive model using the Non Return to Zero (NRZ) signaling proposed in [21,23]. We use this method to design an asynchronous arbiter. Desai [23] also gives various approaches for implementing the bus based interconnect structure. These have been found to result in excessive usage of asynchronous combinational elements to eliminate or cancel residual events arising out of previous register transfers. We give an alternate bus structure based upon the bundled data transfer model which is more efficient in terms of additional asynchronous logic blocks.

2.1 Asynchronous Arbiter

Arbitration circuits are commonly used in implementing digital systems where a single resource needs to be allocated to different concurrent register transfer processes in a behavioral description of the system [17,18,19]. Typical examples are system with shared buses, multiport memories, packet routers etc. An asynchronous arbiter is defined as a circuit that dynamically allocates a single shared resource to the user components in a system. Each user, when it requires the resource, issues an asynchronous request and waits until the arbiter produces a grant. The user on obtaining this grant acquires the resource, uses it to perform certain task and then releases the resource by releasing the request for it. The user can request for the resource again by issuing another request.

The arbiter, on receiving a number of active requests from different users concurrently, issues a grant to exactly one of them after a finite delay and leaves other requests pending until the recipient user has released its request. The arbiter then releases the grant; if there are pending requests, it produces another active grant, again on a mutually exclusive basic. Figure 2.1 represents the block diagram of an N bit arbiter. ***Req1, Req2, .. , ReqN*** are N requests coming from the N different users and the arbiter circuit on receiving these requests issues N active grants on a mutually exclusive basis.

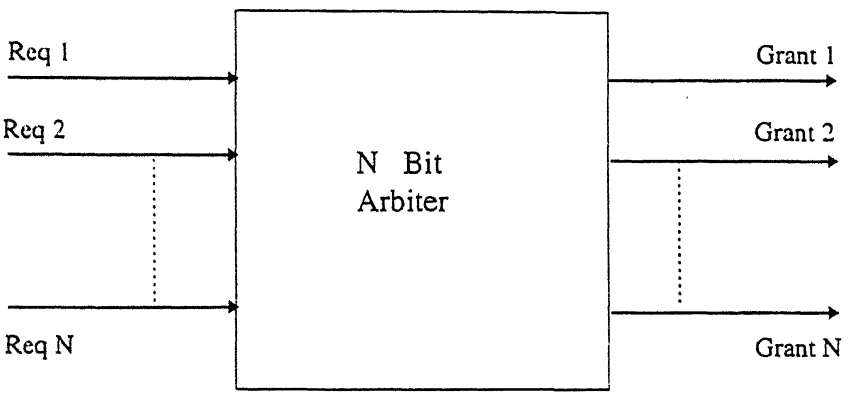


Fig 2.1 Block Diagram of 'N' Bit Arbiter Circuit

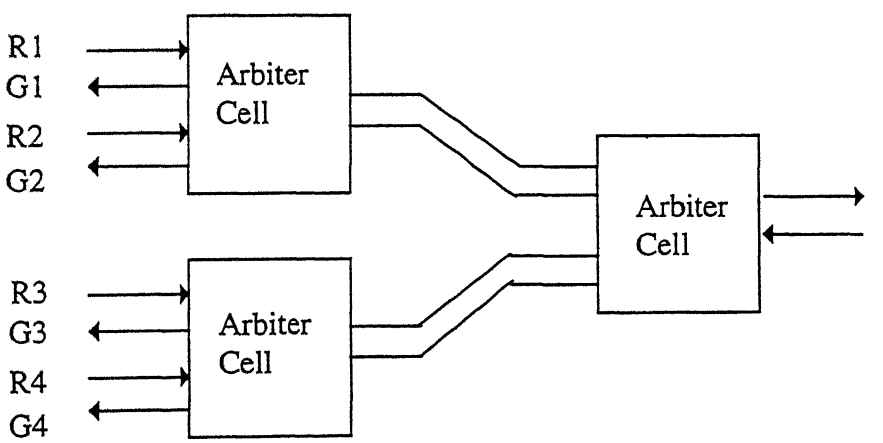


Fig 2.2 Multiway Arbitration

We consider a standard basic cell, called arbiter cell, of a multiway arbiter, that arbitrates between two users. Multiway arbitration is organized by building a cascade of such cells to form a tree structure. Each cell thus propagates the request in the direction from the lower level to higher level of structure, while the grants are generated in opposite direction. Figure 2.2 illustrates the way a cascaded multiway arbiter can be composed from the tree arbiter (TA).

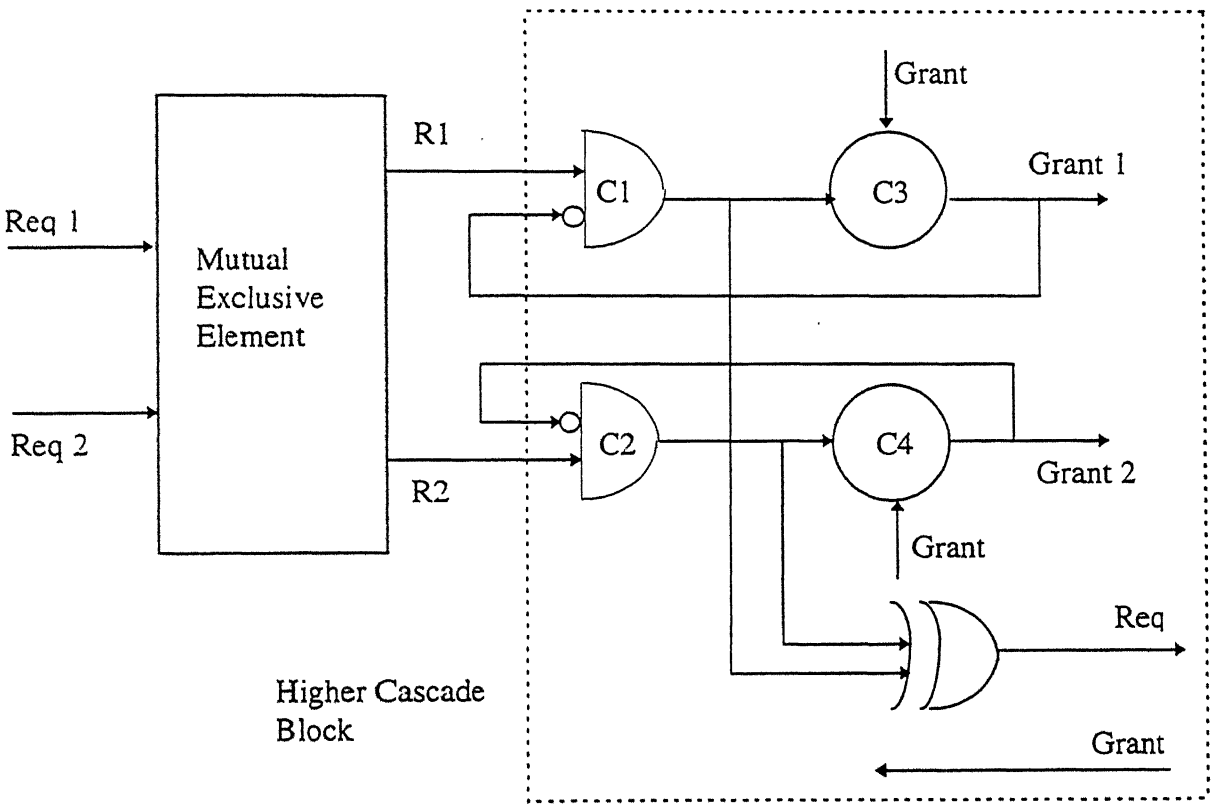


Fig 2.3 Tree Arbiter Circuit

Figure 2.3 gives the functional representation and circuit diagram of a two way tree arbiter circuit (TA). This arbiter circuit uses the event driven two phase non return to zero protocol, where both the control signals and the data are transitions. These transitions can be either a 0 to 1 transition or a 1 to 0 transition.

Figure 2.4 represent the handshake protocol used in the arbitration process. It shows three request-grant handshake links (R1,G1),(R2,G2) and (R,G), where (R1,G1)

and (R2,G2) stand for the links with lower level cascades and (R,G) pair is the link for the higher level cascade.

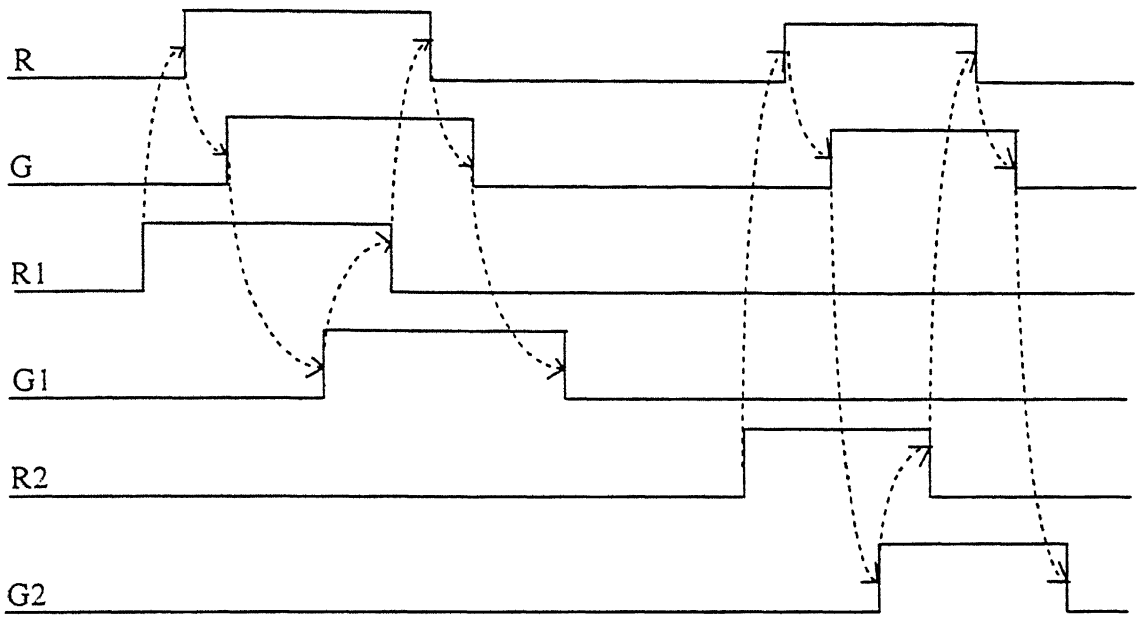


Fig 2.4 Handshaking Protocol

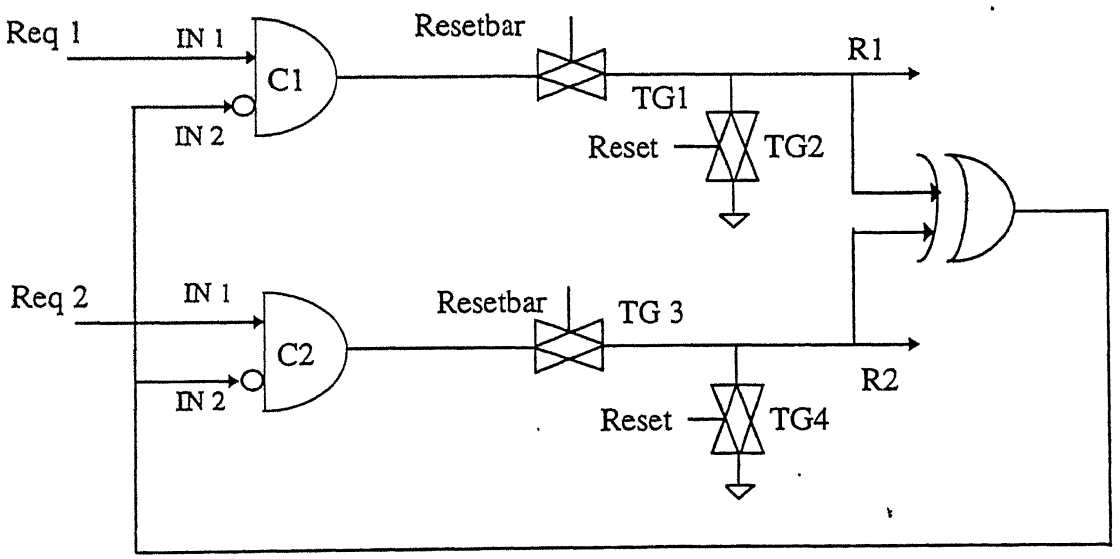


Fig 2.5 Mutual Exclusive Element

The tree arbiter circuit can be divided into two blocks namely the mutual exclusive block (ME) and the higher cascade block (CB). The mutual exclusive element accepts the two requests, which may be overlapping, and generates the nonoverlapping requests. It accepts the request signal ***Req1***, ***Req2*** and generates two non overlapping control signals ***R1*** and ***R2***. These control signals are fed to higher cascade block. CB receives the grant signal from the higher level of cascade and accordingly it generates two grant signals ***Grant1*** and ***Grant2***. The circuit diagram of the mutual exclusive element is shown in Figure 2.5.

2.1.1 Circuit Operation

Each of circuit element in an asynchronous system needs to be taken to a known initial state on power on and global reset. Here it is assumed that on the application of reset, the outputs of all the C elements are set to a logic value '0'. In order to achieve this, a resettable C element is used. When a '0' to '1' transition is applied at the noninverting input of invertible C element. Because of the inversion of one of the inputs of C1 during a power on, or a global reset a positive going transition will exists at this input. Hence C1 will fire and a transition will be placed at the output of C1. This transition will be transmitted through the XOR gate and will finally be placed on IN2 inputs of both C1 and C2. Now if a transition arrives at IN1 of C2 then C2 will not fire. But if a new transition arrives on IN1 of Ci, C1 will fire resulting in a canceling of the transition at IN2 of C2, after which C2 can fire, thus resulting in non overlapping outputs.

The transitions from C1 and C2 will pass through the XOR gate and hence there will be a transition on the ***Req*** line. Once a transition appears on the ***Grant*** line then C3 will fire resulting in a transition on the ***Grant1*** line. It results in a transition residing at the ***Grant*** input of C4, which may result, in future, in an erroneous operation and hence, needs to be cancelled. When the next transition occurs on IN1 of C1, C1 will fire; a transition on ***Req*** line will arrive and corresponding grant will be issued resulting in a transition on ***Grant1***. The next ***Grant*** transition will cancel the extra transition residing on C4. So after servicing the first request the arbiter will now service the second request in a similar fashion. Now we can apply the next set of requests.

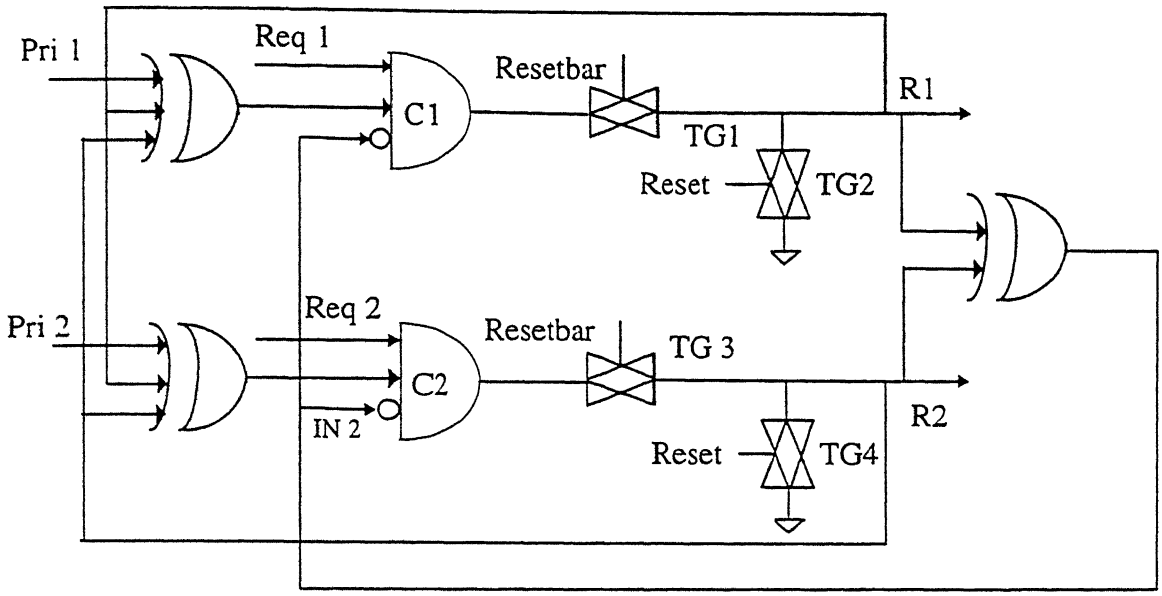


Fig 2.6 Mutual Exclusive Element with Priority

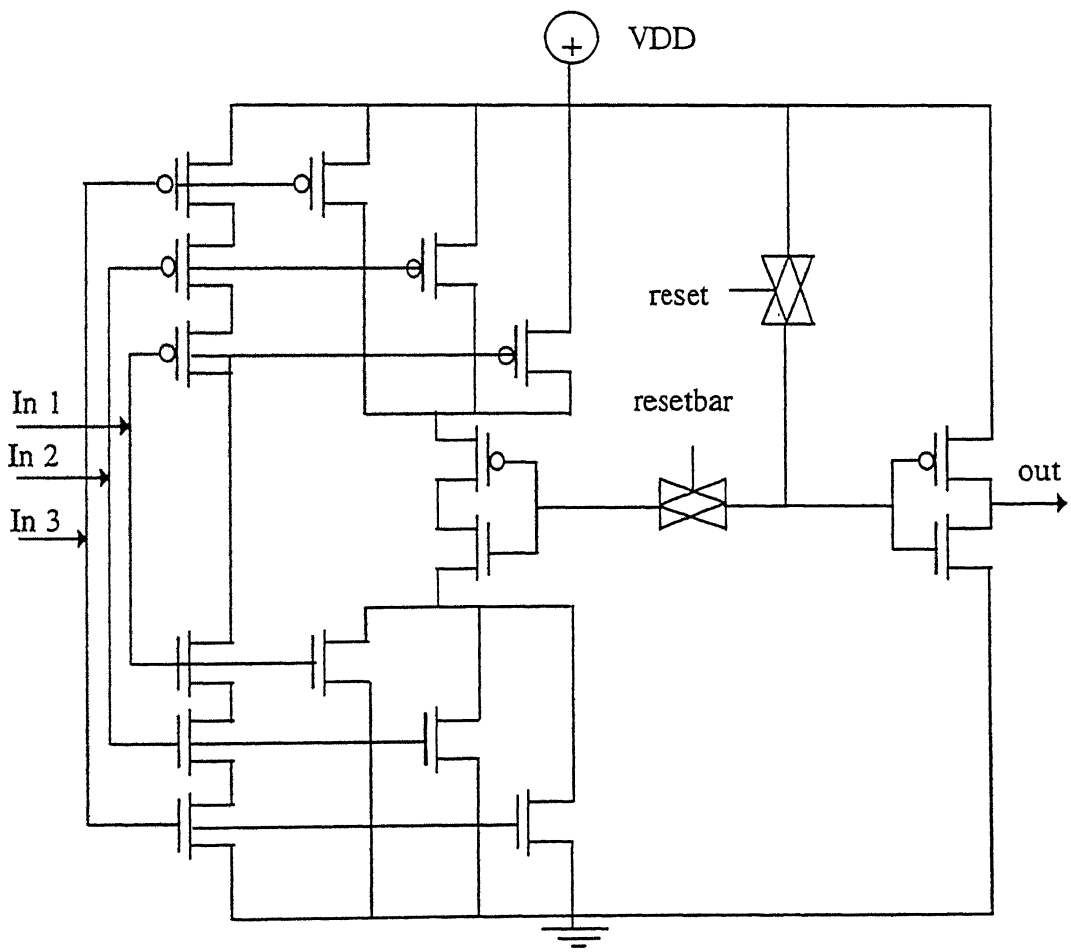


Fig 2.7 Muller C element with three inputs

Another variation to the above mentioned circuit is possible by assigning the priority control signals ***Pri1*** and ***Pri2*** to the tree arbiter. When a transition arrives on ***Pri1***, ***Req1*** is serviced and if transition occurs on the ***Pri2***, then ***Req2*** is serviced. This is shown in Figure 2.6. C1 and C2 shown in the figure are three input Muller C element, the structure of which is as given in Figure 2.7. Whenever, there is a transition on ***Req1*** and ***Req2*** rail simultaneously, then depending on the transition on ***Pri1*** or ***Pri2*** rail, either C1 or C2 will fire and will place the transition on the output of the corresponding C element.

2.2 Bus Architecture with Bundled Data Transfer

2.2.1 Introduction

In order to design digital systems, different resources such as datapath elements, registers, input ports, output ports etc., need to be interconnected using buses. Based upon the intended behavior of the system, the procedure for controlling the movement of data between registers and the logic units through these buses need to be developed. Two ways of interconnecting the datapath elements, namely, the point to point interconnection and bus topology, have been widely used in synchronous systems. We explore the possibility of employing similar interconnection topology in asynchronous digital systems. Desai [23] gives different implementations of the bus interconnection topology. In the point to point interconnection topology, output of one resource is directly connected to the input of another resource. Due to resource sharing, in the case of point to point interconnection topology, time division multiplexing is done, i.e., assigning the resource at different points of time. However, in case of bus topology, the output of each resource is transferred to a common bus, which is usually shared by all resources. The output is then routed to the required destination. Usually the bus transfer is preferred over the point to point interconnection topology because in point to point interconnection topology, the controller tends to become very complex. Also the time taken is more because if the data is to be transferred to more than one resource then direct interconnection is not possible.

2.2.2 Bus Transfer

In the bus transfer topology, output from the datapath elements are transferred to the common bus. Here parallel data transfer is considered. The approach used for bus transfer is based on the two phase bundled data convention. In two phase bundled data convention, if a sender and a receiver communicate using transition signaling, there will be two control wires and many data wires between them as shown in Figure 2.8.

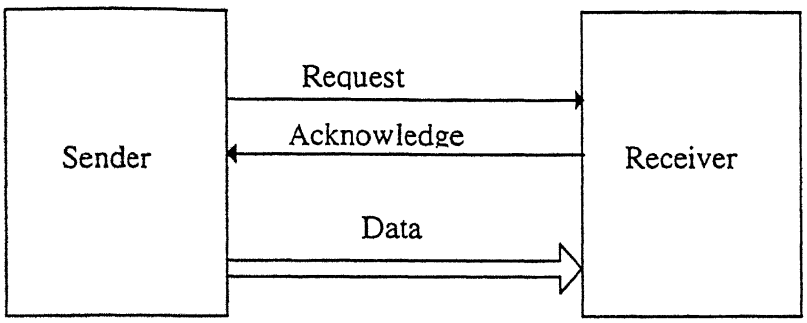


Fig 2.8 Bundled Data Interface

The data wire carry conventional low or high status to convey true or false Boolean values. Sender places a data value on the data wires and then produce an event on its control wire, called ***Request***, to indicate that valid data is available. Receiver then accepts the data and produces an event on its control line called ***Aknowledge*** to indicate that the earlier data has been accepted and that the bus is ready to accept a new data. The three events, ***Data change***, ***Request*** and the ***Acknowledge*** events always follows in the cyclic order and are shown in Figure 2.9.

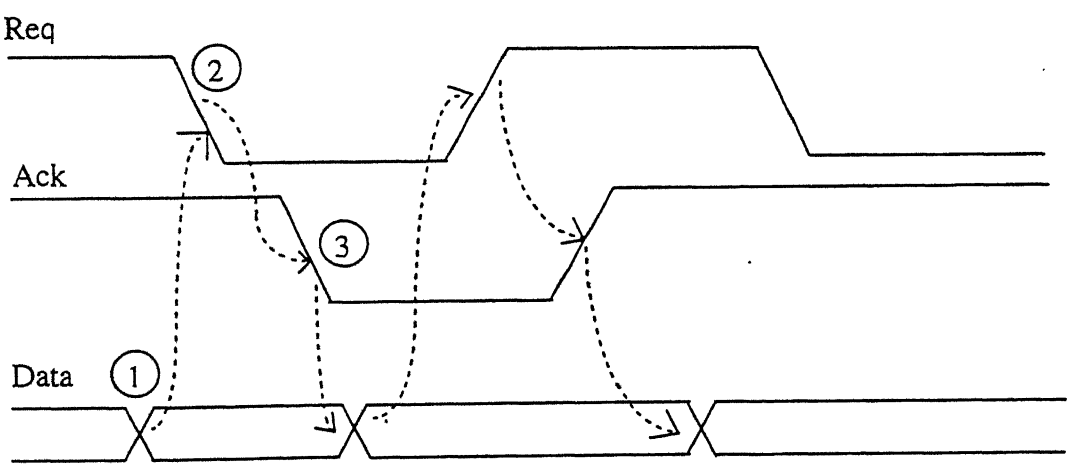


Fig 2.9 Two Phase Bundled Data Convention

Assume that the following resources are available for bus transfer.

- A register file having N registers, each "m" bit wide
- p_i input ports each "m" bit wide
- p_o output ports each "m" bit wide
- A "m" bit wide bus i.e. "m" single rails

In bus structure the following data transfer is possible

- Register to register
- Register to output port
- Input port to register

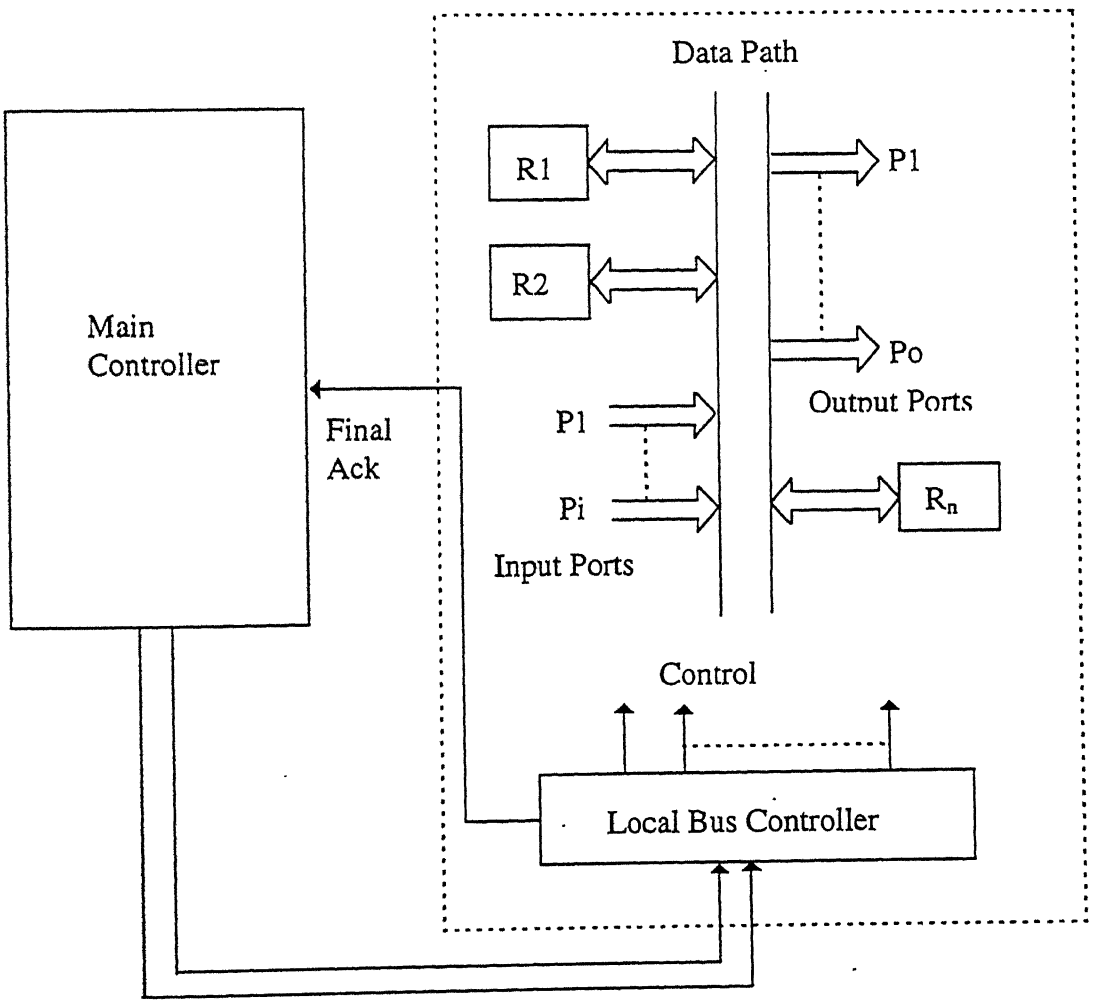


Fig 2.10 Block Diagram of the Bus Structure

In the design methodology adopted, any register communicates with the data path elements in the same way as it communicates with the ports. In the typical bus structure shown in the Figure 2.10, there are *N* registers, ***po*** output ports and ***pi*** input ports, which are connected together through the common bus. The data transfer between them is controlled by the local controller, which on receiving the control signals from the main controller, generates local control signals to coordinate the various activities in the data transfer. Finally a ***Final Ack*** signal is generated to indicate the data transfer and it is fed back to the main controller so that next set of data transfer can be carried out. The block diagram of the general bus interconnection topology structure is given in Figure 2.10.

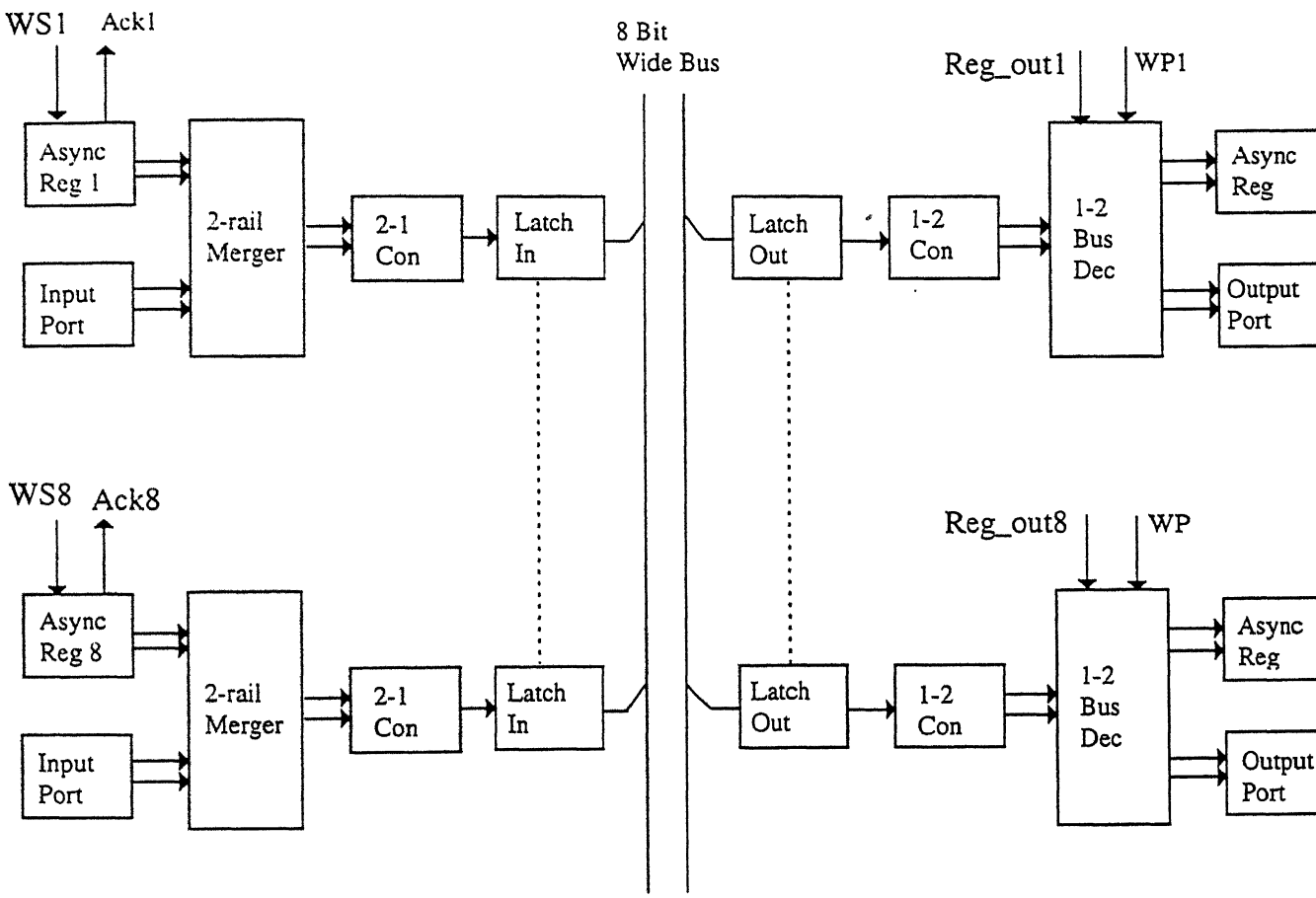


Fig 2.11 Bundled data Transfer Bus Architecture

This bus interconnection topology is entirely different from the schemes proposed by Desai [23]. The interconnection topology is based on the bundled data transfer. Here

we get an improvement in the bus width by reducing it to half. Also there is an improvement in the design of the decoder by way of its reduced complexity.

The main controller provides the following control signals to the local bus controller.

1 For "N" registers
(a) N control wires WS_1 to WS_n to identify the source registers.
(b) N control wires WD_1 to WD_n to identify the destination registers.

2 Register to port transfer control signal, WP_1 to WP_o.
3 A port to register transfer control signal *P_2_r_tr*.

We also assume that for register to register data transfer, transitions are available on WS_i and WD_j. In case of register to port transfer, the transitions are available on WS_i and WP_o and in case of input port to register transfer, transitions are available on WD_i and *P_2_r_tr*. Two distinct phases usually completes the data transfer, viz.,

- removal of stale/past data
- transfer fo desired data.

The bus architecture for a 8 bit bundled data transfer is shown in Figure 2.11

The 2 rail NRZ data from the **Async Register** [23] output and from the input port is merged using a **2 rail merger**. These inputs are merged such that only one of them is routed through the bus. The merger is realized as shown in Figure 2.12 The output of the merger is given to a **2_rail_to_1** converter. Depending on the transition on either rail 1 (r_1), or on rail 0 (r_0), the **2_rail_to_1** converter will give either logic 1 or logic 0 as the output respectively. The structure of a **2_rail_to_1** converter is the same as given in [22] and is shown in Figure 2.13. The output of the **2_rail_to_1** converter is fed to the two phase transition sensitive **latch**, where a transition on the capture input (***C*** and ***CBar***) latches the input data and a transition on pass input (***P*** and ***Pbar***) makes the latch transparent allowing the data to be available on the output. The transition latch is realized as shown in the Figure 2.14

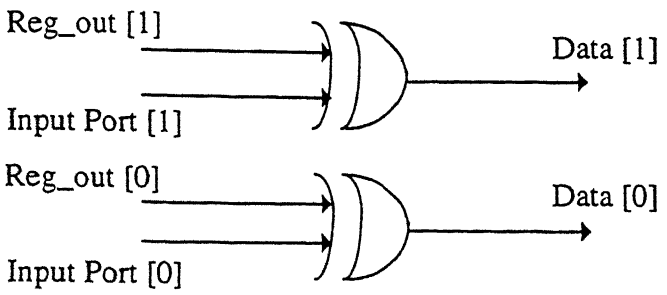


Fig 2.12 Two Rail Merger

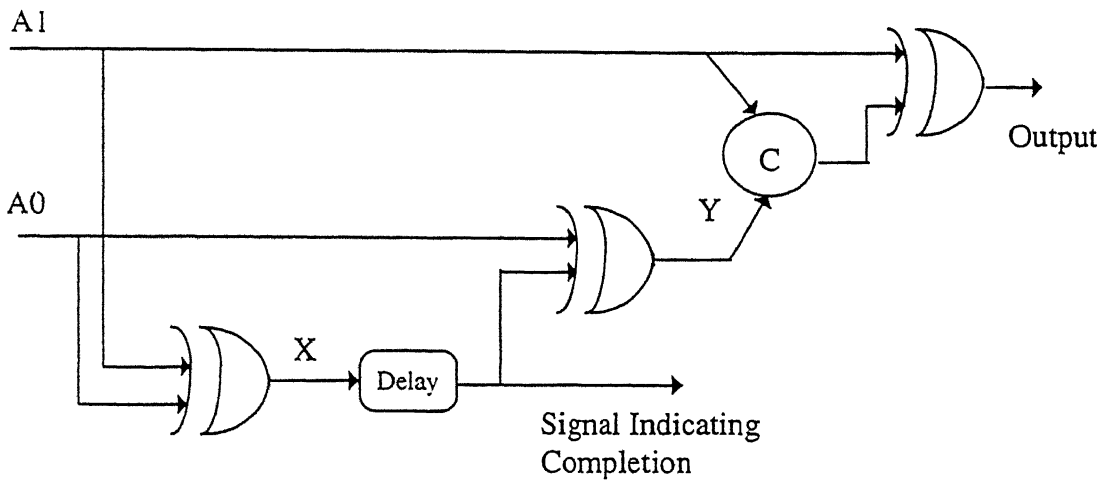


Fig 2.13 2_to_1 converter

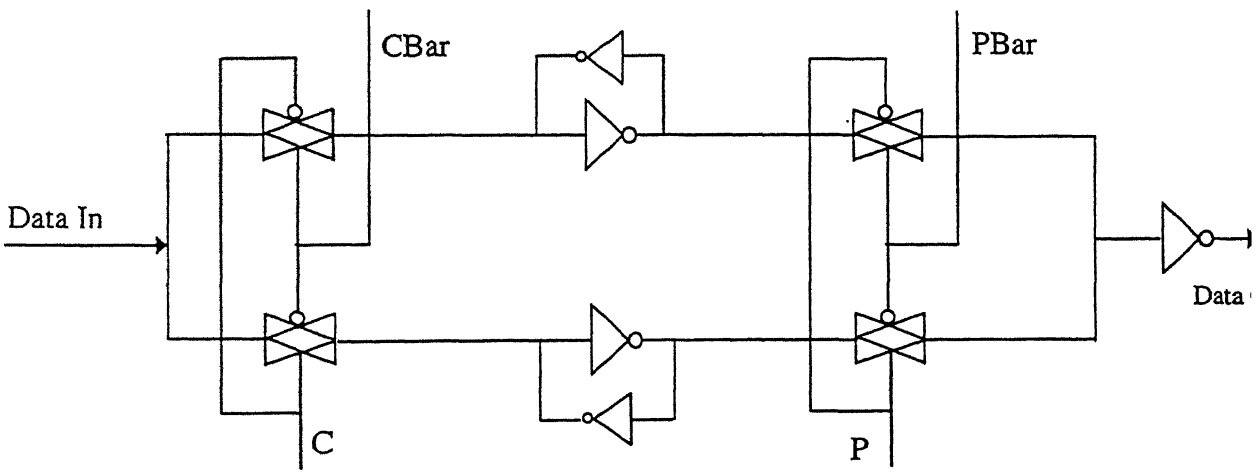


Fig 2.14 Transition Sensitive Latch

Similarly the data from the bus is latched and once the transition arrives on the pass input of the latch, level sensitive Boolean data is passed to a **1_to_2_rail** converter [21]. A **1_to_2_rail** converter, converts the level sensitive Boolean data to the two rail NRZ event data format and generates a transition on either rail 1 (r_1) or on rail 0 (r_0) depending on the value of the data at input being a 1 and 0 respectively. This two rail data is then fed to a bus decoder, which on receiving the control signals corresponding to the desired data transfer will initiate the transfer to its desired destination. Figure 2.15 gives the structure of the **bus decoder**. The bus decoder has an input ***Data_In*** in the two rail format. ***Reg_out*** and ***Port_out*** are the two control signals. The application of one of the control signal results in transfer of input data to the appropriate output and a ***Transf_ack*** signal to indicate the completion of data transfer. Thus transfer of the 2 rail data to a relevent asynchronous register or output port initiates the generation of the ***Transf_ack*** signal as shown in Figure 2.15. These ***Transf_Ack*** signals are merged using **single rail merger** to initiate ***Final_ack*** signal. The signal ***Final_ack*** is sent to main controller to initiate next data transfer.

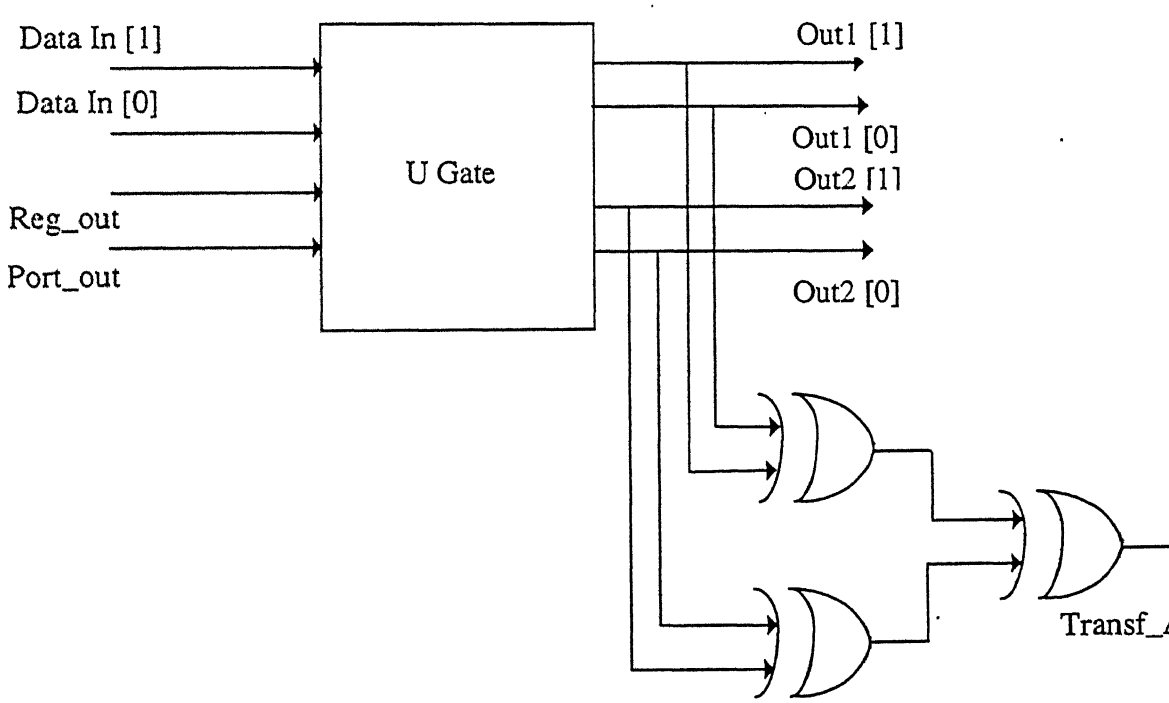


Fig 2.15 Bus Decoder

The structure of the local bus controller is shown in Figure 2.16. Control signal WS_i is applied to the ***data_out*** rail of the source register (**Rs**) and WD_i is applied to destination register (**Rd**). It initiates two concurrent process, viz, placing the data on the ***R_out*** rails of **Rs** and clearing the old data from the **Rd**. When data is placed on the ***R_out*** rail an corresponding acknowledge signal ***Ack*** is generated. This ***Ack*** signal and ***P_2_r_tr*** input signal are passed through a **one rail merger** to distinguish the register to register data transfer and the port to register transfer and also at the same time initiates the process of capturing the data by placing a transition on the capture input of the transition sensitive input latch . The signal indicating the completion of data conversion in a **2_rail_to_1** converter is given to the pass input of the input latch and to the capture input of output latch. The pass transition is generated after passing the capture transition through a delay element. These delays are determined by T_{PLH} (The propagation delay for a low to high transition on the output) and T_{PHL} (The propagation delay for a high to low transition on the output) of the load.

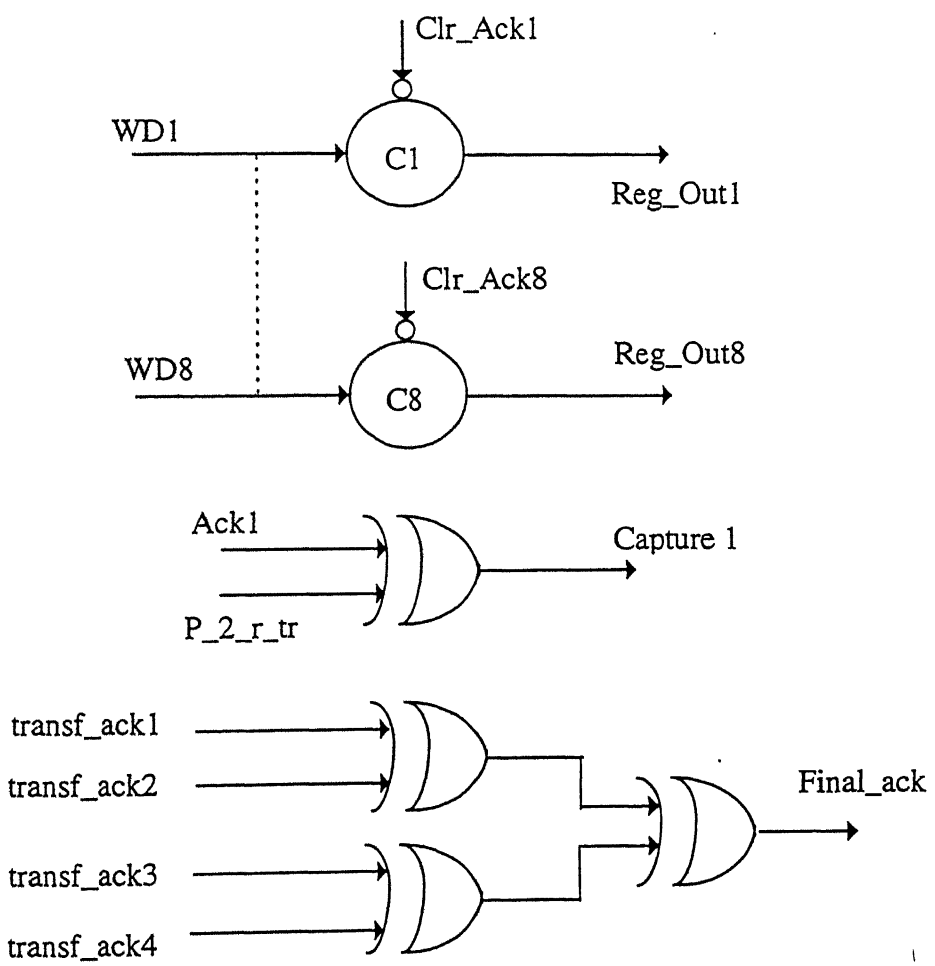


Fig 2.16 Local Bus Controller

As the pass transition arrives on the pass input of the output latch, data is available at the input of **1_to_2_rail** converter, where it is again converted into transitions. Control signal WD_i is given to a Muller C element with an inverted input to generate the ***Reg_out*** control signal, which is connected to the control input of bus decoder. The other control input to the bus decoder is WP_i. Depending on the presence of a transition on ***Reg_out*** or WP_i the data is transferred to either **Rd** or an output port. For 8 bit wide bus, the local controller contains 8 C elements, 8 XOR gates for **one rail merger** and 16 XOR gates for **two rail merger**.

This approach of bundled data transfer has several advantages compared to the approach given in [23]. The main advantage of this approach is that the bus width is reduced by half. In a complex system such as a microprocessor most of the silicon area is consumed by the bus, hence by reducing the bus width a significant amount of Silicon area can be saved. Another advantage is the design of the bus decoder. Here the design of bus decoder is very simple as there is only one feedback compare to 15 feed backs in [23], which complicates the design of the data decoder. However, in this approach, determining the appropriate values of the delays to generate the pass transitions for input and output latch is one of the most important design issues. Implicitly, it also results in the implementtation being not truly delay insensitive.

Chapter 3

Synthesis of Asynchronous Digital circuits

3.1 Introduction

One of the main motivation of this work is to synthesize asynchronous digital circuits from hardware description language (HDL) such as Verilog or VHDL. We assume that the circuit will be designed using our approach. The logic and layout designers convert each functional block into a logic or schematic, which is captured by schematic capture tools and simulated to verify its functionality, timing and fault coverage. One can apply, the describe and synthesize methodology, at several levels of abstraction. At the gate level, functional unit and control unit logic can be synthesized by means of conventional combinational logic synthesis. Controllers can be synthesized by means of finite state machine (FSM) by sequential synthesis. At the register transfer level (RTL), behavior is described with program algorithms, flowcharts, data flow graphs, instruction sets or generalized FSMs in which each state performs arbitrarily complex computations.

High level synthesis (HLS) is a sequence of tasks that transform a behavioral representation into an RTL description [24,25,28]. The design consists of employing functional units, such as ALUs, multipliers, etc.; storage units, such as memories and register files; interconnection units, such as multiplexer and buses. A generic high level synthesis system is shown in Figure 3.1.

The compiler converts the behavioral description into an internal representation. The RTL library contains the physical and simulation models of components to be used during synthesis. The main advantage of HLS are productivity gains by moving the design process to a higher level of abstraction. The automation provided by HLS ensures a more systematic and efficient search of the large design spaces created by the shift to a higher level of abstraction.

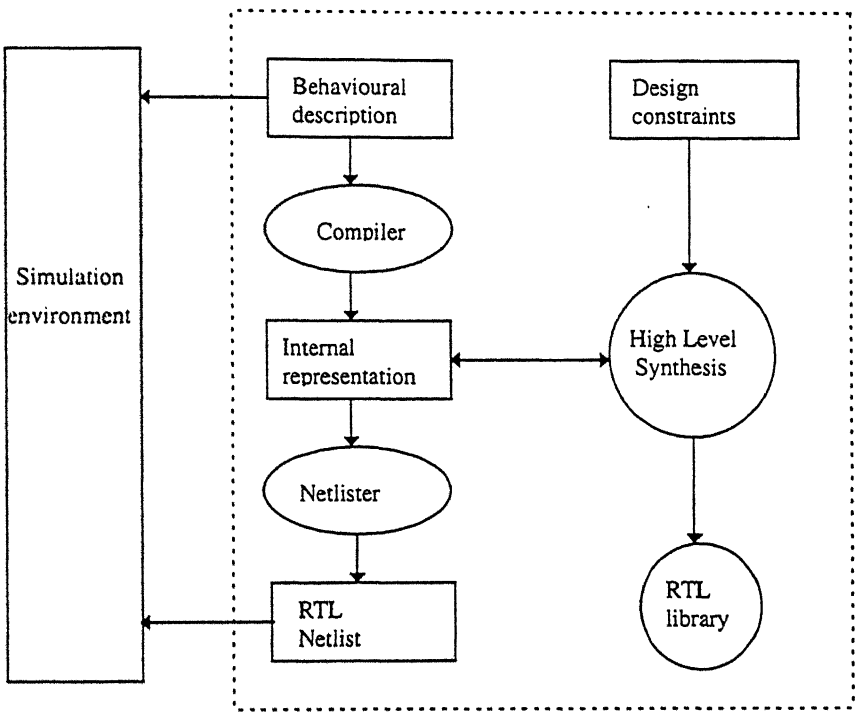


Fig 3.1 A generic high level synthesis system

The input description specifies the intent of design. It is usually an algorithmic description of the design behavior that does not contain structural information such as component types and their interconnections. Nor does it provides information about circuit structure, such as number of pipeline stages or clock phases.

The HLS system compiles the behavioral description into an internal representation [28]. All synthesis tasks work from this representation. There are several types of internal representation. The most convenient type of representation is the one that matches the problem more closely. For example, for a digital filter which repeatedly performs a series of operations on an infinite input data stream, one needs to represent

the data, arithmetic operations and read and write dependencies that defines the order of execution. A data flow graph (DFG) representation is an ideal choice. Consider the following expression

$$Y = max((A\ shr\ 1) + (B - (B\ shr\ 3)), B)$$

Here "shr" represents the shifter. The data flow graph of the above expression is shown in Figure 3.2. Here operations O_1 and O_2 can be executed in parallel.

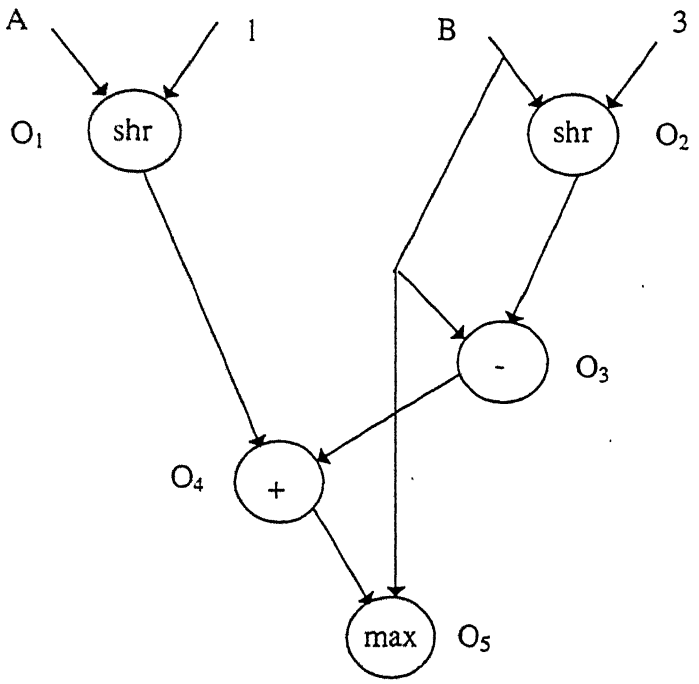


Fig 3.2 DFG Representation

DFG consists of a set of nodes, each node representing one operation in the original description. Two nodes o_i and o_j are connected by an arc if there is a data dependency between them, i.e. the result of operation o_i is an input to o_j. In other words dependency arc connecting o_i and o_j indicates that the operation o_j cannot execute before o_i executes. Since DFG is based on data dependency alone it is the most parallel representation of the description. However, a DFG is not sufficient to represent reactive or embedded systems, in which the control sequence is based on external conditions. For such systems we must represent the control flow in addition to data dependencies. This is achieved by augmenting the DFG with control nodes. The augmented representation style

is called as control data flow graph (CDFG), which allows control constructs such as conditional branches and loops.

3.1.1 HLS Model

Logic synthesis is based on the formalism of Boolean algebra, where as sequential synthesis is based on the FSM model. For HLS the FSM model is extended by incorporating variable assignment. One such model is known as Finite state machine with datapath (FSMD) model. It can represent all classes of synchronous hardware design and can be used for representing both control dominated and data dominated systems. A control dominated design consists of a large control unit and a small data path, e.g. serial to parallel converter. A data dominated design such as FIR filter has minimum control and a large data path to perform filtering operations.

3.1.2 Synthesis Task

HLS maps a behavioral description into the FSMD model so that the data path executes variable assignments and the control unit implements the control constructs. Essentially the synthesis process consists of four distinct phases [25].

- Scheduling
- Resource sharing and binding
- Interconnection and storage allocation
- Control unit generation

Efforts on HLS have been mainly focused on synchronous design. Most of the existing scheduling algorithms [24,27] are based on the concept of the control step, i.e. time is measured in cycles and the cycle time is determined by the worst case delay of all operations executed in a control step. From the point of view of the timing model, since time is considered as a continuous variable, and initiation and completion of operation are events that can occur at any instant, we need scheduling strategies to synthesize asynchronous circuits. Furthermore, any asynchronous circuit implementing a design can be represented using two distinct blocks; controller and the datapath which interact as shown in Figure 3.3

As shown in the Figure 3.3, the control is completely distributed in such a way that for each data path block (or a group of data path blocks), there is a local controller (LC) [23]. A local controller has two types of handshaking signals. These are the local signals and global signals. Local signals are the request and completion signals for synchronization within the data path block being controlled and other signals such as, operation code for ALUs or the selection code for MUX. Global signals are for the synchronization associated with data transfer between blocks. Without the loss of generality we assume that a LC exists for each computational block and the input MUXs.

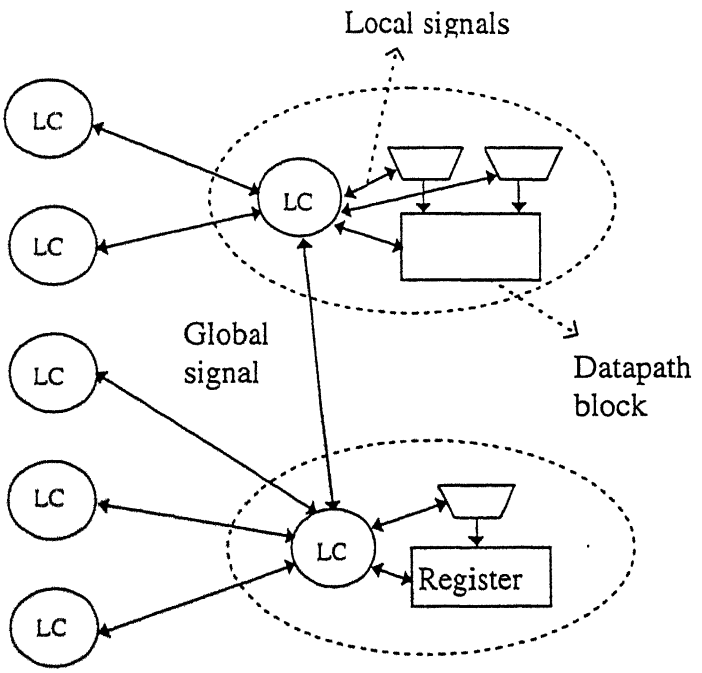


Fig 3.3 Synthesized design organisation

The execution of an operation is performed in three steps. These are

- The input data is read from the register
- Operation is executed in a FU.
- Output data is written into a register

Here we assume that the delays associated with a read operation on a register and a write operation into the register is data independent and that these delays can be included in the delays of each individual operation.

In this chapter we describe the two scheduling algorithms. The first algorithm known as event list scheduling (ELS) [14] is based on the popular list based scheduling algorithm used in the synthesis of synchronous systems. The second algorithm is based on our extension of the first algorithm. These algorithms perform the operation scheduling and hardware allocation of the asynchronous digital circuits.

3.2 Scheduling

Scheduling problem is defined with a DFG, G(V,E) where vertices V and edges E denotes operations and the dependencies, respectively. For each vertex $v \in V$ we define the following

v_o : executed operation
v_s, v_c : start and completion time (after the node being scheduled)
f_{vo} : functional unit that executes operation v_o
pred(v) = {u | (u,v)∈E}
succ(u) = {v | (u,v)∈E}

Besides the functional behavior of any datapath element, a library of functional units (FUs) is available for a given technology and is represented by the delay matrix (DM) as shown in Table 3.1. With each element in the delay matrix we associate a delay ($d_{f,o}$) for the execution of an operation o by the FU type f, $d_{f,o}=\infty$ implies that o is not implemented by FU f. Beside this we can also associate an additional parameter like the area of silicon with each element in the delay matrix. FU(o) denotes the set of FU types implementing operation o.

	Delay(ns)				
Functional Unit	+	-	<	*	/
Adder	35	∞	∞	∞	∞
Mul	∞	∞	∞	85	∞
ALU	50	50	85	∞	∞
DIV	∞	∞	∞	∞	100

Table 3.1 Delay Matrix (DM)

A resource vector, $\mathbf{R} = < |f_1| , |f_2| , \ldots\ldots\ldots , |f_n| >$, represents a set of resources available for allocation, where $|f_i|$ indicates the number of instances of the FU f_i. Here we assume that all $|f_i| > 0$ (if $|f_i| = 0$ then f_i can be eliminated from the library and is not available for allocation). Given a resource vector, the average delay, $\delta_{o,avg}$ for the operation o can be defined as

$$\delta_{o,avg} = \frac{\sum_{f_i \in FU(o)} |f_i| \, \delta_{f,o}}{\sum_{f_i \in FU(o)} |f_i|}$$

$\delta_{o,avg}$ is the pre-scheduled estimate of the expected execution delay for operation o by assuming that there is an equal probability that any vertex can be assigned to any $f \in FU(vo)$.

Given a DFG G(V,E), a library of FUs represented by the delay matrix DM, and a set of resources **R** our aim is to find an asynchronous schedule for G. Scheduling algorithms can be classified into two major classes based upon the constraint that they assume.

1. ***Resource Constrained Scheduling***: Resource constrained scheduling problem is encountered in many applications where we are limited by the silicon area. The constraint is usually given in terms of number of FU. Here the schedule is gradually constructed, one operation at a time, so that resource constraints are not exceeded and data dependencies are not violated. Since in the design methodology adopted, the circuit implementation works correctly irrespective of the delays in the various blocks, the worst case delay is not required to be associated to ensure the proper functioning. The scheduling is done by assigning the average case delay. Using the average case delay increases the possibility of a better schedule.

2. ***Time constrained scheduling***: Time constraint scheduling is important for designs targeted towards applications in a real time system, e.g., in a DSP application, sampling rate of input data stream dictates the maximum time allowed for carrying out a DSP algorithm on the present data stream, before the next data stream arrives. Here our main objective is not only to reduce the cost of hardware, but at the same time the design should work under all possible execution delays. Thus, worst case delay needs to be

considered for each FU. Besides this, the delay in interconnection network and in the controller should be accounted for.

Finding an asynchronous schedule means determining a partial order of the vertices and allocating each operation to a type of FU so that the total estimated processing delay of the schedule is minimized. Scheduling a vertex $v \in V$ means defining v_f, v_s, & v_c, i.e the type of FU that will execute v_o, its estimated start and finish times respectively. Allocation selects the number and type of hardware unit, that will compose the data path. In operation scheduling, operation are distributed through the time space and are assigned to a type of FU. Scheduled data flow graph (SDFG) is a modification of the CDFG containing information relating to scheduling and allocation. The result of the scheduling on DFG is represented by a SDFG as shown in the Figure 3.4.

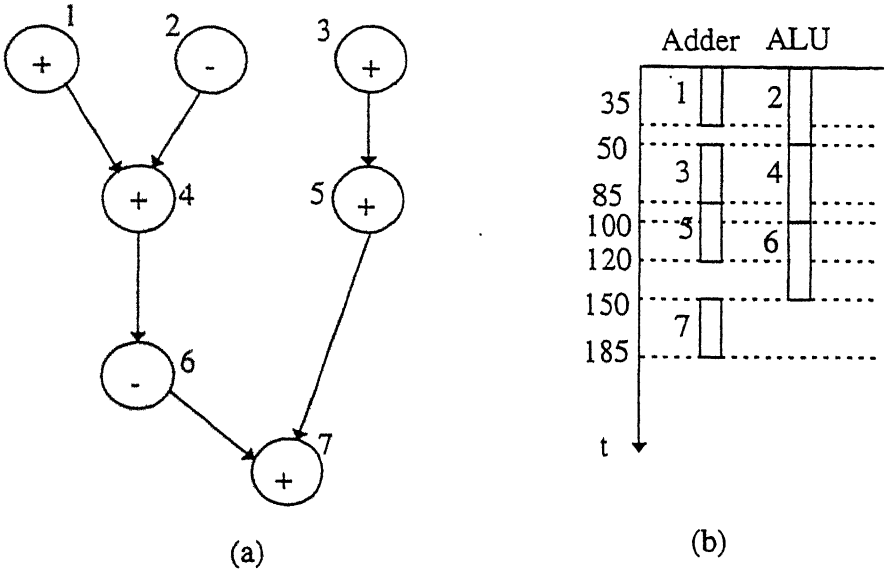


Fig 3.4 (a) DFG (b) SDFG {resources <1 ALU, 1 Adder>}

In the above example we assume one adder and one ALU. The numbers near the vertical bars represent the operations. Since ALU can perform both addition and subtraction so operation 1 and 2 are scheduled simultaneously and allocated to the adder and the ALU respectively.

3.2.1 Data Structure

The algorithm employs the frame reservation table ($\mathbf{FRT_f}$) [14] as the data structure bound to a type of FU **f**. $\mathbf{FRT_f}$ indicates the number of resources of a particular type **f** that are active at each time instant. $\mathbf{FRT_f}$ is updated each time a new operation is scheduled and bound to a FU of type **f**. However, the number of active instance at any time instant can never be greater than |f|.

The frame reservation table $\mathbf{FRT_f}$ is represented as the event list ($\mathbf{EL_f}$) The event list $\mathbf{EL_f}$ contains two members $< \mathbf{time_i}, \mathbf{nfus_i} >$ ordered by time, where, $\mathbf{nfus_i}$ represents the total no of available FU of type **f** which are non active, from time **I** to time **I+1**. At the start, since all the functional units are non active, so each $\mathbf{EL_f}$ contains one event $< \mathbf{0}$, |f| >. Fig 3.5 depicts the $\mathbf{FRT_f}$ and the $\mathbf{EL_f}$ corresponding to the scheduled graph shown earlier in the Figure 3.4.

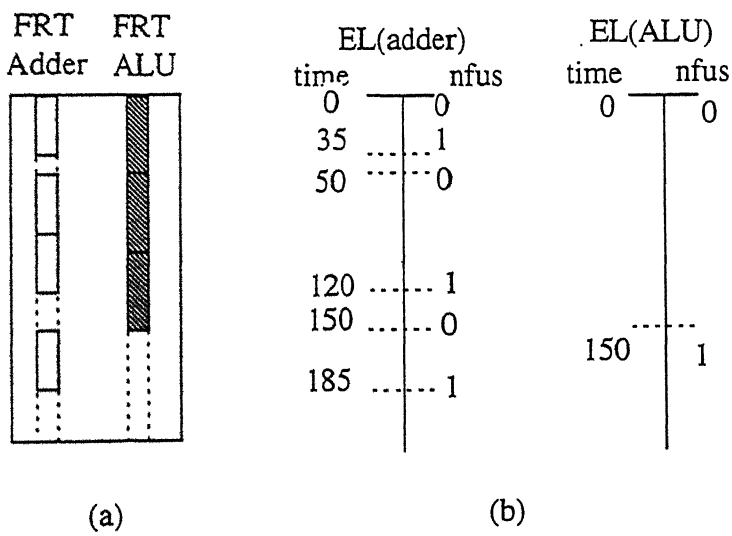


Fig 3.5 (a) Frame Reservation Table (b) Event List

The event list scheduling algorithm (ELS), uses two functions to manage the event list. They are *find_free_interval* and the *reserve_interval.*

3.2.2 Find_free_interval

This function is defined as

$$min\_start\_time = find\_free\_interval\ (EL_f,\ start\_time\ ,\ delay)$$

This function takes the ***start_time*** and the ***delay*** parameter as inputs and checks in the event list $\mathbf{EL_f}$ of a particular functional unit type **f**, for the first time interval of duration delay with $min\_start\_time \geq start\_time$, such atleast one functional unit of type **f** is available for the duration delay . This is shown in the figure 3.6

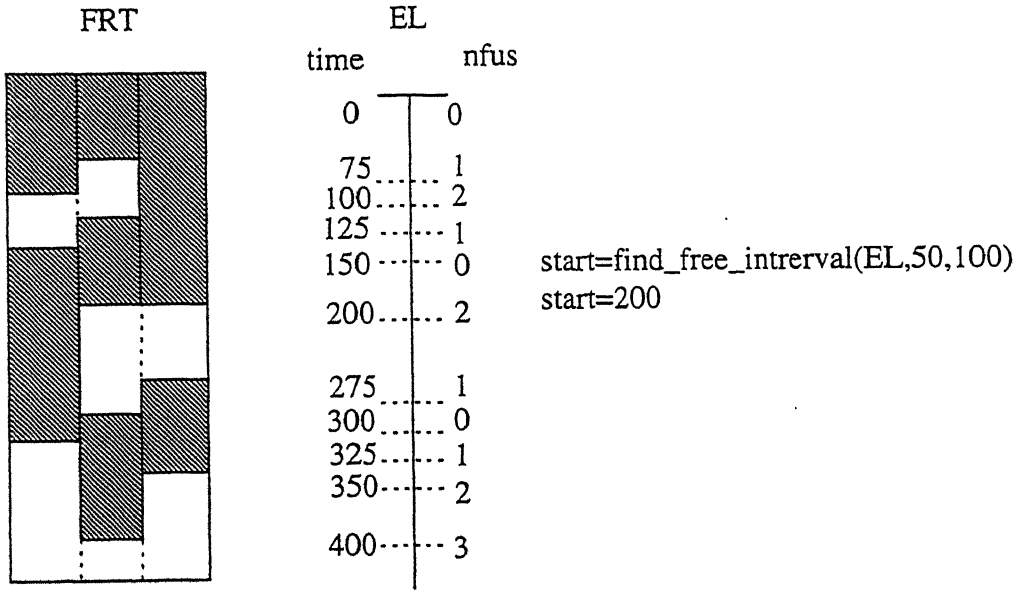


Fig 3.6 Execution of Function Find_free_interval

3.2.3 Reserve_interval

This function is defined as

$$EL_f = reserve\_interval(EL_f,\ min\_start\_time\ ,\ delay)$$

This function accepts the ***min_start_time*** as computed by the function ***find_free_interval,*** and the ***delay*** and reserves the time interval corresponding to the

duration delay starting at ***min_start_time*** in an event list of a particular FU, **f**. So the event list gets modified after reserving this time interval. It is shown in figure 3.7

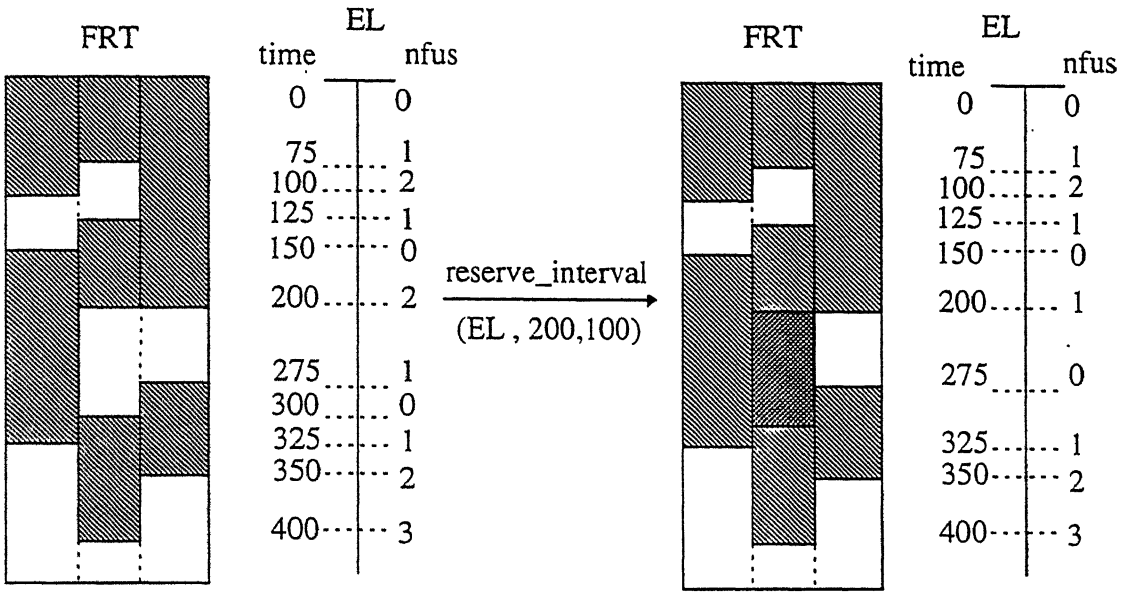


Fig 3.7 Effect of Function Reserve_interval on an Event List

3.3 Scheduling Algorithms

We next describe the two algorithms for scheduling operations in an asynchronous system. As mentioned earlier these algorithms are similar to the list based scheduling employed for the resource constrained scheduling in synchronous systems. List based scheduling algorithm [24,25] maintains a priority list of ready nodes. During each iteration the operations in the beginning of the ready list are scheduled till the resources are exhausted in that state. The priority list is always sorted with respect to. a priority function. Scheduling an operation may make some other non-ready operations ready. These are inserted into the list according to the priority function. These iteration continues till all the lists are empty.

Similar to the list based scheduling algorithm both the algorithms associate a priority function with every node. It is this priority function which resolves the resource contention among ready operations. There are various types of priority functions. Any one of these can be used as a priority function in scheduling; such as, mobility range, length of the longest path, number of immediate successors, delay along the path length

till the terminal node. The quality of schedule greatly depends on the type of priority function chosen.

The two algorithms differs in the way the priority function is calculated. In the case of event list scheduling algorithm the priority function is calculated in the beginning; where as, in the case of modified event list scheduling algorithm the priority function is dynamically recomputed by looking for those ready nodes having the same priority, which are of same type, and which have the same data dependencies with a common node at some lower level of the DFG.

During the execution of the scheduling algorithm the set of the nodes of the DFG are partitioned into three sets, the ready set (*RS*), the scheduled set (*SS*) and non scheduled set (*NSS*). *SS* contains all those vertices which are already scheduled. A vertex belongs to the *RS* if it is not yet scheduled but all its predecessors have already been scheduled. Rest of the vertices belong to the *NSS*. The scheduling algorithm is described as given in Figure 3.8.

```
Event_list_scheduling(G(V,E), R, delay_matrix)
{
   for each type of FU f, initialize event_list (EL_f , lfl );
   calculate_priority_function ( G, R, delay_matrix);
  RS=find_source_vertices(G);
  SS={ϕ};
  NSS=V-RS;
  while RS ≠ NULL
  {
    v=max_priority_vertex(RS);
    min_start=  max  (u_c);
                 u∈pred(v)
    for each f∈ FU(v_o)
      {
       min_start_time_f = find_free_interval( EL_f , min_start, δ_f,vo);
       completion_f = min_start_time_f + δ_f,vo;
      }
       find that FU f_min such that completion_f_min = min  (completion_f );
                                                       f∈FU(v_o)
```

```
            reserve_interval (EL_fmin , min_start_time_fmin , δ_fmin,vo);
            v_s= min_start_time_fmin;
            v_c= completion_fmin;
            v_f=f_min;
            SS= SS ∪ {v};
            fireable_nodes = {v ∈ NSS | ∀ u ∈ pred(v), u ∈ SS};
            RS = ( RS - {v}) ∪ fireable_nodes;
            NSS =  NSS - fireable_nodes;
        }
    }
```

Fig 3.8 Event List Scheduling Algorithm

Similar to the list scheduling, ELS calculates the priority for each vertex of the DFG. Then vertices in the ready_set are scheduled in an order according to their priority. Path length to the end and the delay along the path length are the priority functions used in this algorithm and is evaluated at the start of the algorithm.

First, all the event lists are initialized with their corresponding number of the resources (lfl), as initially no resource is allocated to any operation. Then the priority of each vertex is calculated and *RS*, *SS*, and the *NSS* are computed. The main loop of the algorithms selects the node with the maximum priority from the *RS*. Then it calculates the completion time for each functional unit that can execute this operation and finds out the FU f_{min} such that it gives the minimum completion time. Finally, v is scheduled and bound to f_{min}. So *SS* is modified with the inclusion of v and v is removed from the *RS*. Now it checks which all nodes becomes ready for scheduling and those nodes are inserted in the *RS* and are removed from the *NSS*. The information about the utilization of the resources is kept in the event list and when an operation can be executed by more than one type of the FU, ELS algorithm tends to bind the vertex to the fastest available FU of the appropriate type.

Event list scheduling uses a greedy strategy to select the next vertex to be scheduled i.e. the one with the highest priority with the FU that completes the earliest. This can lead to inferior results. In modified event list scheduling algorithm the priority of each node is dynamically computed based on the information that the two or more

nodes in the ***RS***, having the same priority, performing the same operation, having the same successors at some level . If this holds then these nodes need to be scheduled first so that their successor node can become ready for being scheduled at an earlier point of time. As will be shown later, this has the potential to reduce the total execution time of the scheduled operations considerably.

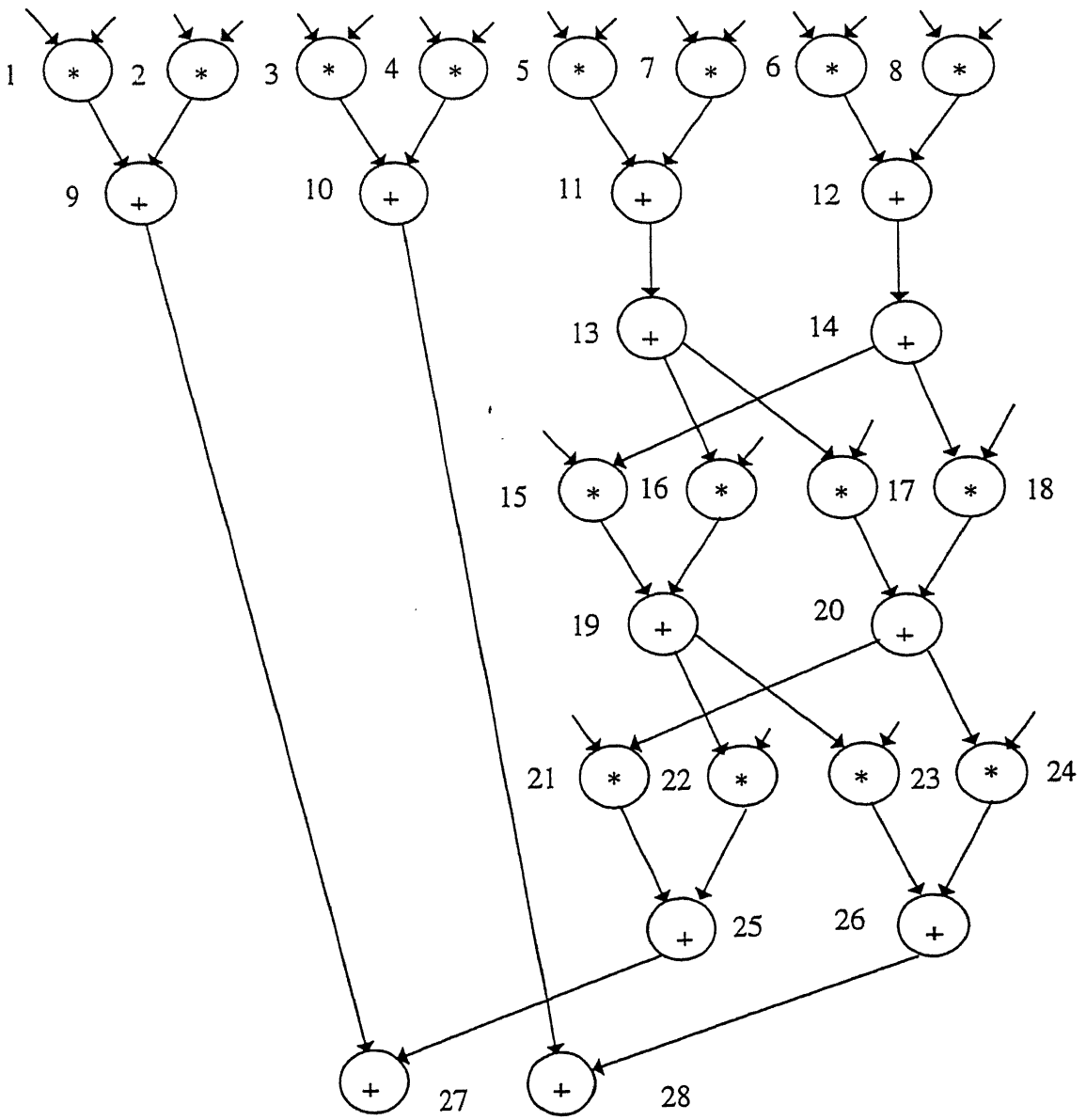
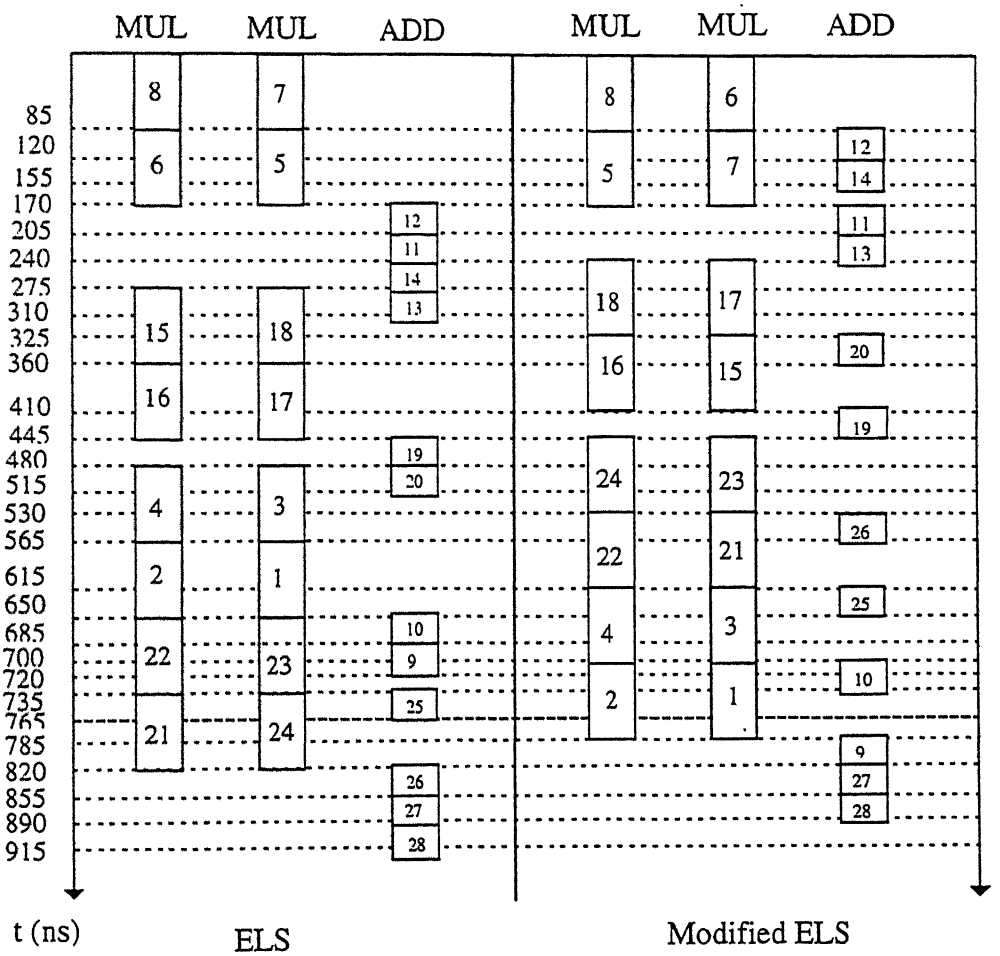


Fig 3.9 DFG of the ARMA Filter

These algorithm have been implemented and tried out on various HLS benchmark examples, such as Differential Equation Solver[24], Fifth-Order Wave Elliptical Filter[14] and AR Filter[27] and Inverse Discrete Cosine Transform (IDCT) [20]. All these examples are scheduled with respect to the delay of the operators given in Table 1.

Figure 3.9 gives the DFG for the ARMA filter and Figure 3.10 gives the schedule for AR filter using both ELS and the modified ELS.



⊗: Multiplier
⊕: Adder
◊: ALU

Fig 3.10 Schedule for AR filter (resources ⊗,⊗,⊕)

It is evident from the above figure that using the ELS algorithm first, operations 8 and 7 are scheduled based upon the greedy strategy. Then operations 6 and 5 are scheduled. The scheduling of the operations 8 and 6, and 7 and 5, will only result in the

operations 12 and 11 to become ready for scheduling. In this case, scheduling of operation 12 is delayed till operation 6 has been scheduled. However, in the modified ELS, operations 8 and 6 are scheduled simultaneously, thereby enabling operations 12 and 14 to become ready earlier. As can be seen this results in an improvement in the total execution time. Table 3.2 depicts the result obtained for the AR filter under different resource constraints.

Resources	ELS (time)	Modified ELS (time)
⊗,⊗,⊕	925 ns	890 ns
⊗,⊗,⊗,⊗,⊕,⊕	480 ns	480 ns

Table 3.2 Result of the AR filter

It is evident from Table 3.2 that when there are only two multipliers and one adder, the modified ELS algorithm gives better results. When the restriction on the number of the resources is removed, both the algorithm give identical results.

The second example used is that of a Differential Equation Solver, the behavioral description of which is given below. Figure 3.11 represents the data flow graph (DFG) of the differential equation solver.

```
while (x<a)
{
  x1 = x + dx;
  u1=u+ 3xudx - 3ydx
  y1=y+udx
  x=x1
  y=y1;
  u=u1;
}
```

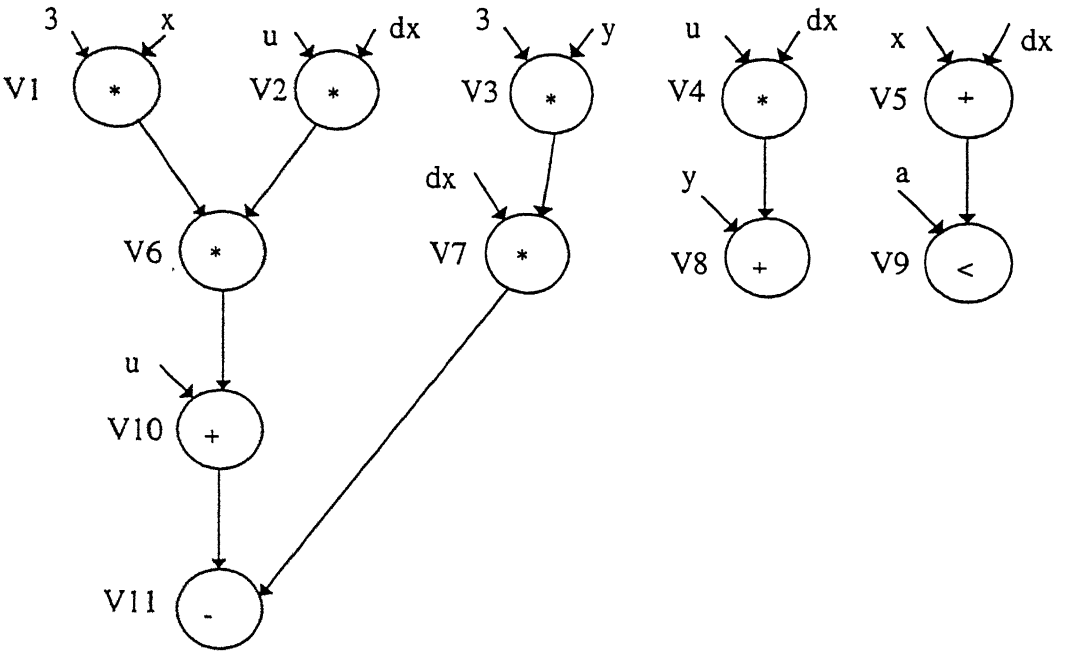


Fig 3.11 DFG of the Differential Equation Solver

Figure 3.12 depicts the resulting scheduling and Table 3.3 presents the results obtained for the differential equation solver under different resource constraints.

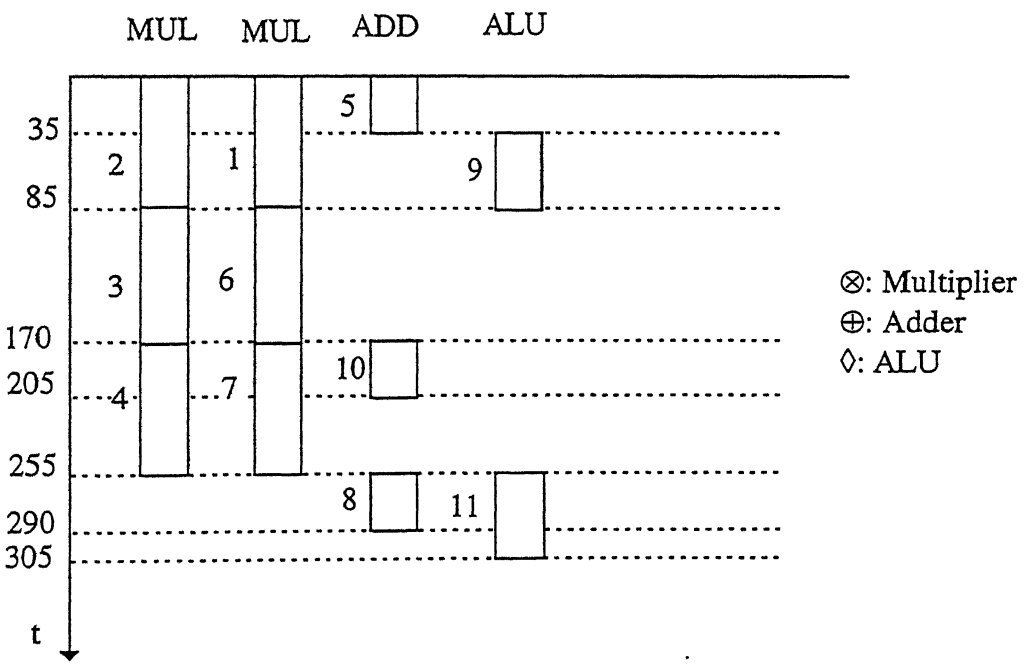


Fig 3.12 Schedule for differential Equation Solver (Resource ⊗,⊗,⊕,◊)

Resources	ELS (time)	Modified ELS (time)
⊗,⊗,⊗,⊕, ◊	270 ns	270ns
⊗,⊗,⊕, ◊	305 ns	305 ns
⊗,⊕, ◊	545 ns	545 ns

Table 3.3 Results of the Diffential Equation Solver

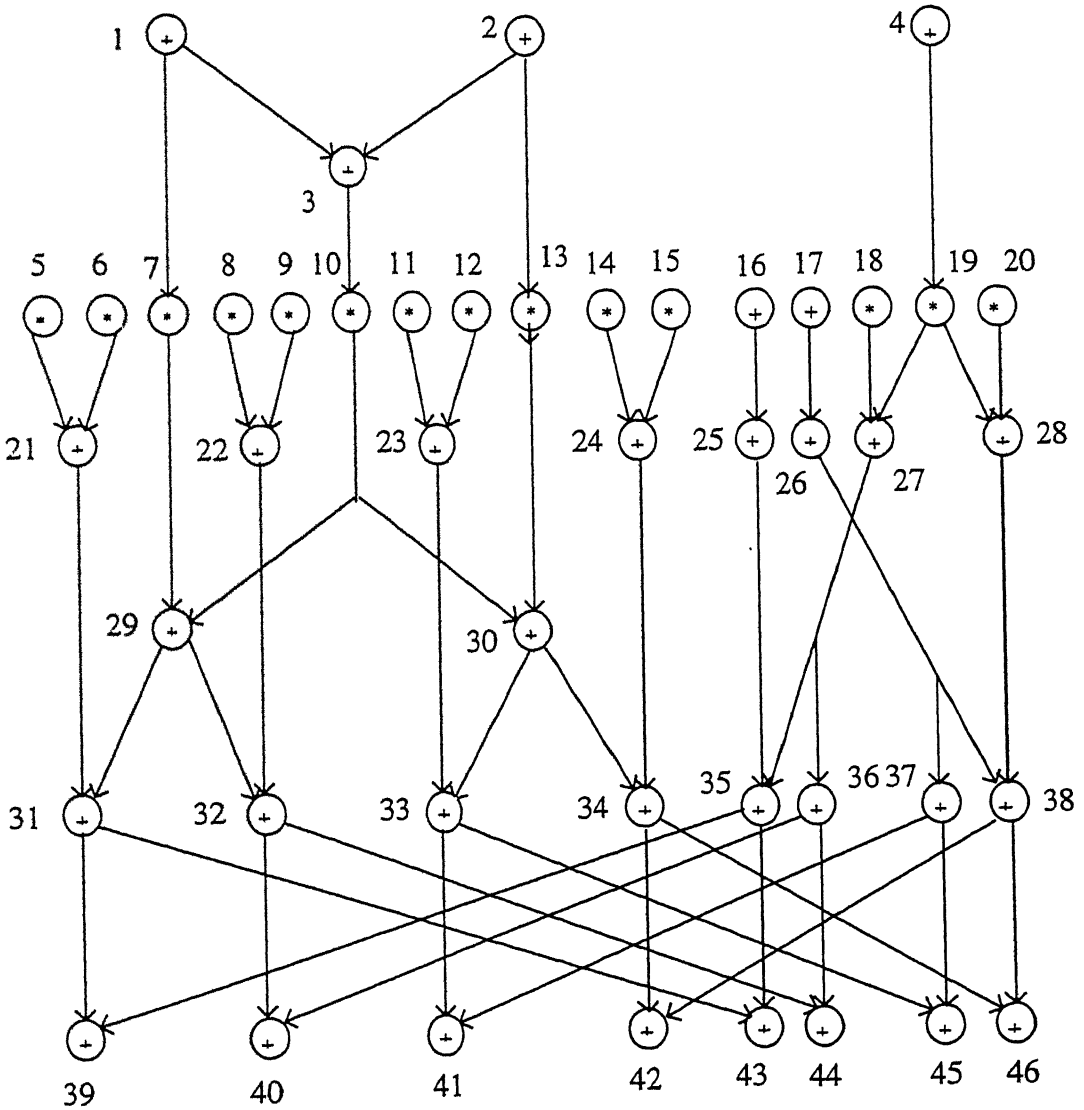


Fig 3.13 DFG of Inverse Discrete Cosine Transform

The third example used for demonstration is an Inverse Discrete Cosine Transform (IDCT), the DFG of which is shown in Figure 3.13. The schedule of the IDCT for three multipliers, three adders and one ALU is shown in Figure 3.14. The results obtained under different resource constraints are given in Table 3.4

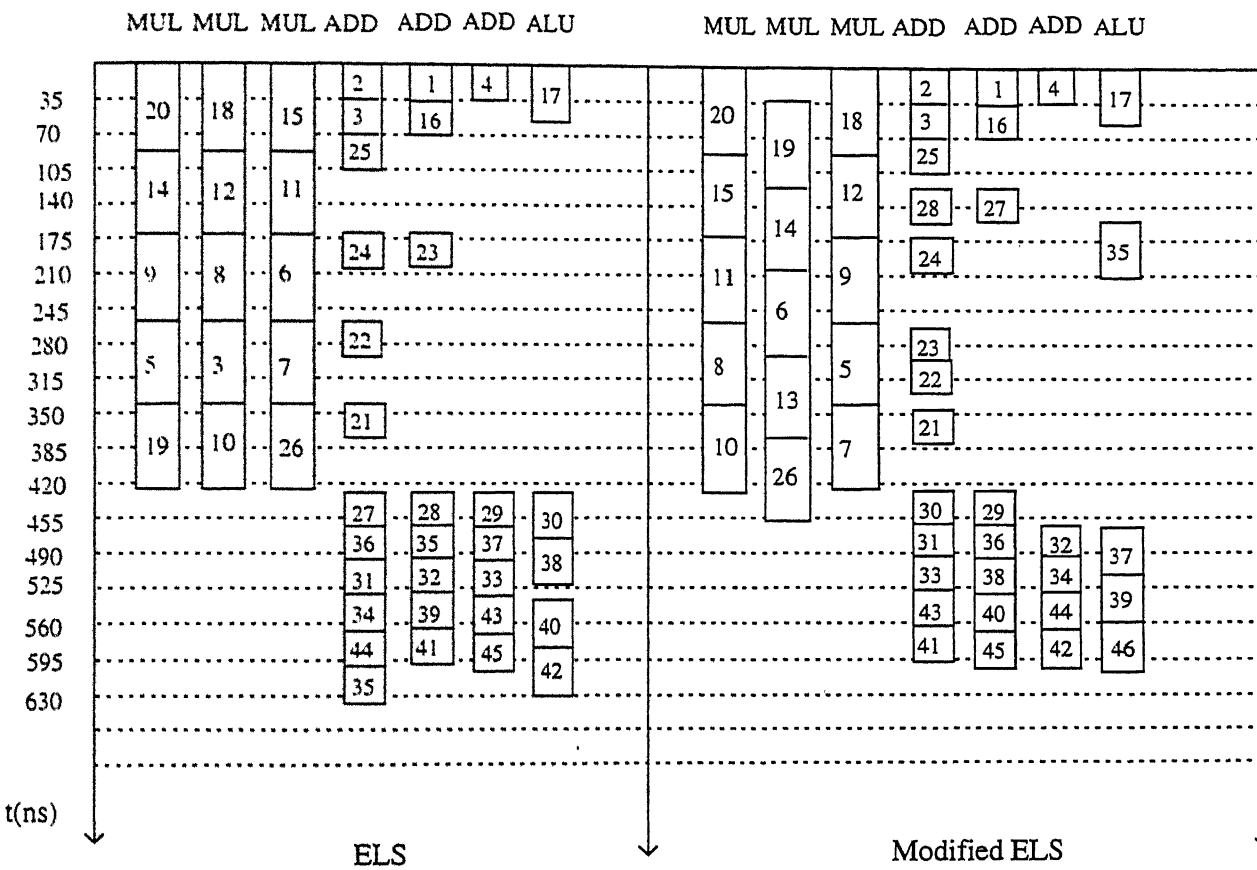


Fig 3.14 Schedule for IDCT (Resources ⊗,⊗,⊗,⊕,⊕,⊕,◊)

The Fourth example used for testing the algorithm is the Fifth Order Wave Elliptical Filter. The DFG of the wave filter is given in Figure 3.15. The result of the application of the ELS and the modified ELS algorithms under different resource constraints are tabulated in the Table 3.5.

Resources	ELS	Modified ELS
⊗,⊗,⊕,⊕,◊,◊	845 ns	830 ns
⊗,⊗,⊕,◊	1030 ns	1015 ns
⊗,⊗,⊗,⊗,⊕,◊	760 ns	740 ns
⊗,⊗,⊗,⊕,◊	845 ns	825 ns
⊗,⊗,⊗,⊕,⊕,⊕,◊	635 ns	610 ns
⊗,⊗,⊗,⊗,⊕,⊕,⊕,◊	550 ns	515 ns
⊗,⊗,⊗,⊗,⊕,◊,◊,◊	590 ns	540 ns
⊗,⊗,⊗,⊗,⊕,⊕,◊,◊	550 ns	540 ns
⊗,⊗,⊗,⊗,⊕,⊕,⊕,◊,◊,◊	495 ns	480 ns
⊗,⊗,⊗,⊗,⊗,⊗,⊕,⊕,◊,◊	465 ns	455 ns

Table 3.4 Result Of IDCT

⊗: Multiplier
⊕: Adder
◊: ALU

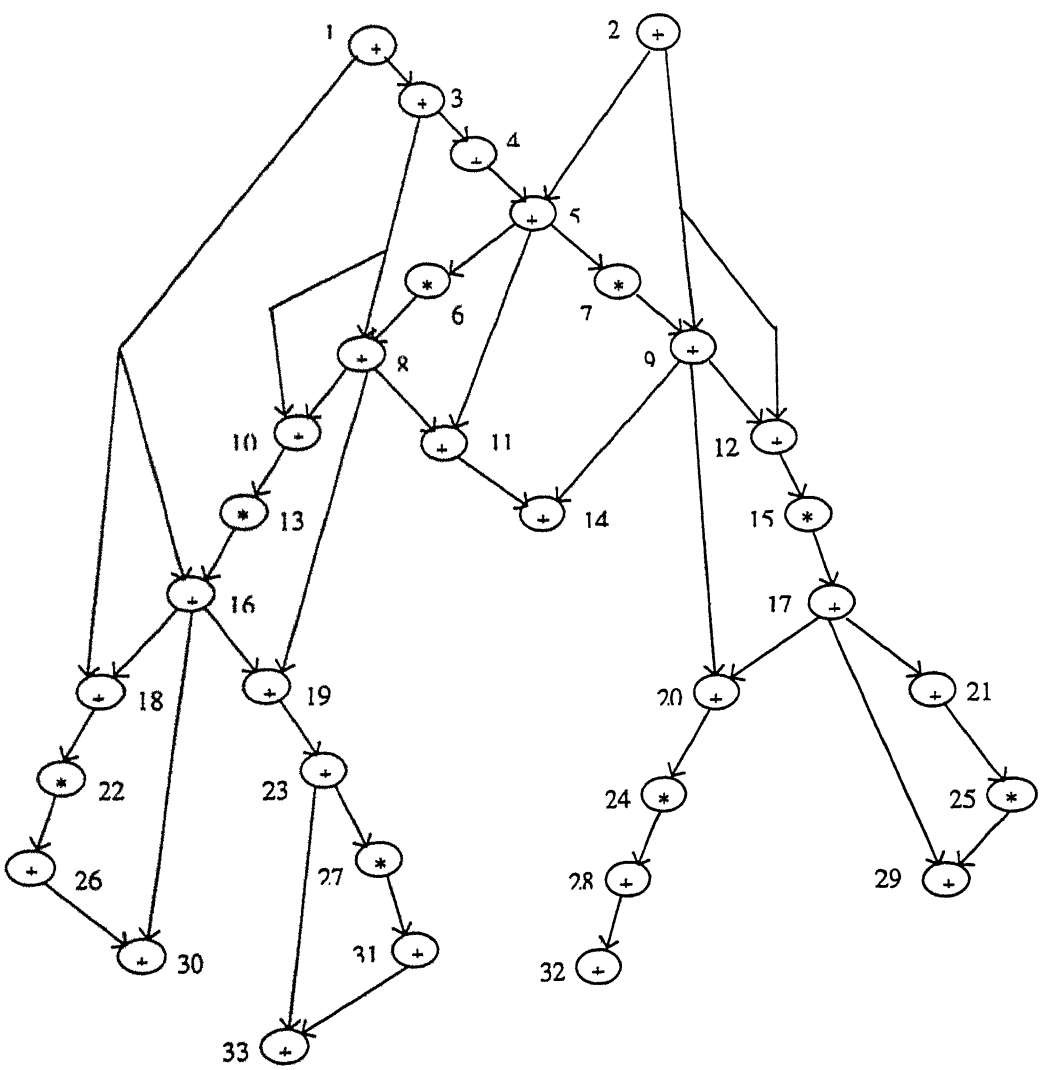


Fig 3.15 DFG of Fifth Order Wave Elliptical Filter

Resources	ELS	Modified ELS
⊗,⊗,⊕,◊,◊	720 ns	720 ns
⊗,⊗,⊕,◊	760 ns	760 ns
⊗,⊗,⊕,⊕,◊	690 ns	690 ns
⊗,⊗,⊗,⊕,◊	745 ns	745 ns
⊗,⊗,⊗,⊕,◊,◊	700 ns	700 ns
⊗,⊗,⊗,⊕,◊,◊,◊	685 ns	685 ns
⊗,⊗,⊗,⊕,⊕,◊	655 ns	655 ns
⊗,⊗,⊗,⊕,⊕,◊,◊	655 ns	655 ns
⊗,⊗,⊕,⊕,⊕,◊	670 ns	670 ns

Table 3.5 Result Of Elliptical Filter

The above results confirms that , if there is no resource contention, in that case modified ELS gives the same results as observed with the ELS algorithm.

Another example used to test these algorithms is a test example [20]. In this example we have input and output nodes and different types of processing units, denoted by A, B and C. In this example, A represents Multipler (*), B represents Adder (+) and C represents ALU (<). Graphical representation of test example is given in Figure 3.16. The results of the application of the above scheduling algorithms are depicted in Table 3.6

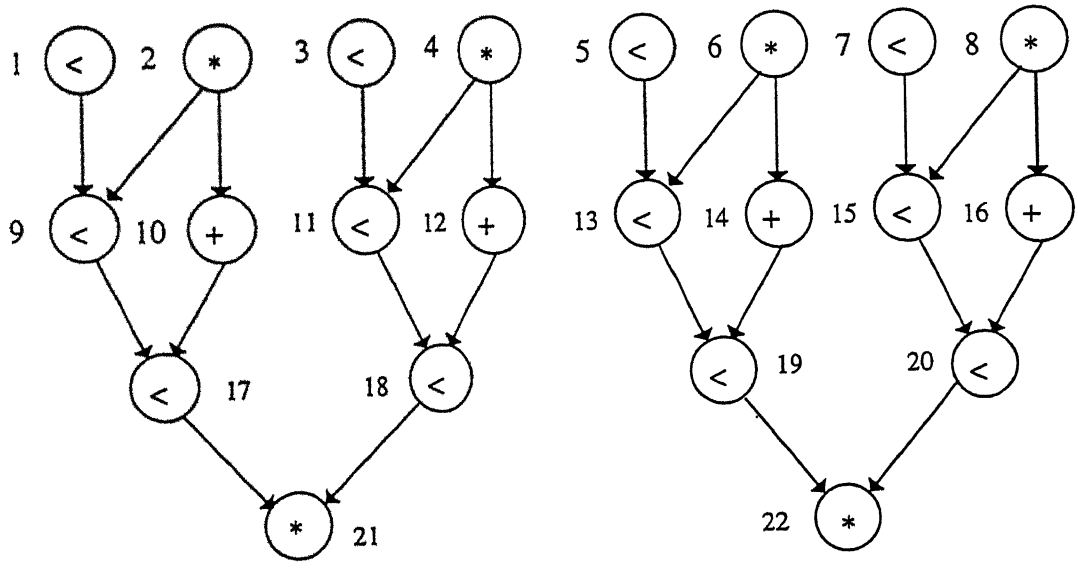


Fig 3.15 Graphical Representation of Test Example

Resources	ELS	Modified ELS
⊗,⊕,◊	720 ns	685 ns
⊗,⊕,⊕,◊	670 ns	585 ns
⊗,⊕,⊕,◊,◊	470 ns	420 ns
⊗,⊗,⊕,⊕,◊,◊	385 ns	385 ns
⊗,⊕,⊕,⊕,◊,◊,◊	370 ns	370 ns
⊗,⊕,⊕,⊕,⊕,◊,◊,◊,◊	320 ns	320 ns

Table 3.6 Result Of Test Example under Resource Constraints

Besides the above examples, these algorithms have been tested on randomly generated DFGs. We first generate undirected random graphs. Every node in the graph is randomly chosen to represent an operator from a given set of predefined operators. The cycles in the graph are first removed by running the depth first search (DFS) algorithm on the random graph to generate a directed acyclic graph (DAG). Then ELS and the modified ELS were employed to schedule these DFGs using the same operator delay matrix as earlier. However, since these are randomly generated DFGs, these do not gaurantee that a severe resource constraint conditions is present. Hence the results obtained by both the algorithms are identical and are summarized in Table 3.6 and Table 3.7.

	Resources		
Nodes	⊗,⊗,⊕,◊,◊	⊗,⊗,⊕,⊕,◊,◊,◊,◊	⊗,⊗,⊗,⊗,⊕,⊕,◊,◊
100	1400 ns	815 ns	1200 ns
200	3235 ns	1625 ns	2350 ns
300	4970 ns	2485 ns	3600 ns
400	6720 ns	3360 ns	4850 ns
500	8430 ns	4270 ns	6100 ns
600	10220 ns	5145 ns	7350 ns
700	11970 ns	5985 ns	8600 ns
1000	17350 ns	8610 ns	12385 ns
2000	34720 ns	17360 ns	24850 ns

Table 3.7 Results obtained using ELS on Random Graphs

	Resources		
Nodes	⊗,⊗,⊕,◊,◊	⊗,⊗,⊕,⊕,◊,◊,◊,◊	⊗,⊗,⊗,⊗,⊕,⊕,◊,◊
100	1400 ns	815 ns	1200 ns
200	3235 ns	1625 ns	2350 ns
300	4970 ns	2485 ns	3600 ns
400	6720 ns	3360 ns	4850 ns
500	8430 ns	4270 ns	6100 ns
600	10220 ns	5145 ns	7350 ns
700	11970 ns	5985 ns	8600 ns
1000	17350 ns	8610 ns	12385 ns
2000	34720 ns	17360 ns	24850 ns

Table 3.8 Results obtained using Modified ELS on Random Graphs

Chapter 4

Implementation of the asynchronous circuits in the Xilinx FPGA

4.1 Introduction

FPGAs are an efficient hardware target when only small number of chips are needed. They are also useful for rapid prototyping. The FPGAs are complex enough to implement more than glue logic. They can implement complex designs up to several thousands gates. As the logic capacity of FPGAs increases, synthesis for FPGAs becomes more and more important.

To efficiently exploit the increased logic capacity of FPGAs, synthesis tools and efficient synthesis methods for FPGA targetting becomes necessary. One solution to designing large designs efficiently is to use hardware description languages (HDL) such as VHDL and Verilog and synthesize it for a target architecture. Several synthesis tools exist for mapping these descriptions to various FPGA families. By using synthesis tools, the modeling, verification and implementation processes can be integrated.

Using a synthesis-based approach, retargeting a design to other technologies becomes possible at little extra cost. Thus, synthesis is attractive for designing chips with small series and for rapid prototyping. When using FPGAs for rapid prototyping, synthesis can be targeted at FPGAs to exercise it for verification purposes, and later an ASIC implementation can be derived. The major advantage of synthesis -based designs is that the same hardware description language code can be used for verification and

implementation. This integrated design flow reduces the design cost, risk of inconsistencies between different modules and the time to the market considerably. Once the functional correctness of the model has been proved, the same code should be usable to generate a hardware implementation. Ideally, this process would require only recompilation with a silicon compiler to yield the final chip.

While ideally, the synthesizable HDL model should be the same for all target technologies, the efficiency of the resulting design is very much dependent on the description and technology used. In order to overcome the optimization problem, look up table (LUT) based architectures (such as the Xilinx FPGAs) are used for efficient implementations. In LUT based architectures, the logic functions are implemented as memory look up tables; therefore, the propagation delay is independent of the function being implemented.

For our work we have used a synthesis tool from Silicon Automation System (SAS). Pinnauq [30.31] translates a RTL description in Verilog to a netlist in the target technology based on the Xilinx FPGA architecture. It supports the XC2000, XC3000, XC4000 and XC5200 series of FPGA from Xilinx. It accepts the description of the design and generates the Xilinx netlist format (XNF) file and an synthesized output Verilog file. This output Verilog file can be tested by applying the same test vectors as applied to the original description for validating the synthesis process. The XNF file generated as a result of synthesis is directly accepted by the Xilinx backend development tool "XACT". XACT maps the results to the target FPGA , using partitioning, placement and routing of the various resources within the FPGA.

In this chapter a brief introduction to the Xilinx XC4000 series of FPGA is given. We use this to implement our asynchronous modules. The synthesized schematics of some of the basic elements, such as, Muller C element, U gate, transition latch etc. are described. Finally some larger asynchronous circuits and systems such as adders, bus decoders, bus arbiters, point to point interconnection topologies and bus architectures for bundled data transfer are synthesized. The synthesis results are reported in terms of the number of FMAPs (mapping logic on F function generator of 4000 series), HMAPs, IOBs (Input output Blocks) etc. used.

4.2 Xilinx XC4000 FPGA

The XC4000 family achieves high speed through improved architecture and supports system clock upto 50MHz. It is a dynamic hardware offering on chip RAM, wide input decoders and has the complexity upto 20000 gates level[ ]. Xilinx FPGA[29] comprises three major configurable elements, configurable logic block (CLB), input output block (IOB) and interconnections. CLB provide the functional elements for constructing the user's logic. The block diagram of CLB is shown in Figure 4.1. The IOB provides the interface between the package pins and the internal signal lines. The programmable interconnect resources provides the routing paths to connect the inputs and outputs of the CLBs and IOBs to the appropriate networks. Besides the above resources, Xilinx FPGAs also consists of on chip ultra fast RAM, dedicated high speed carry propagation circuit, wide edge decoders, internal 3 state bus capability and global low skew clock distribution network.

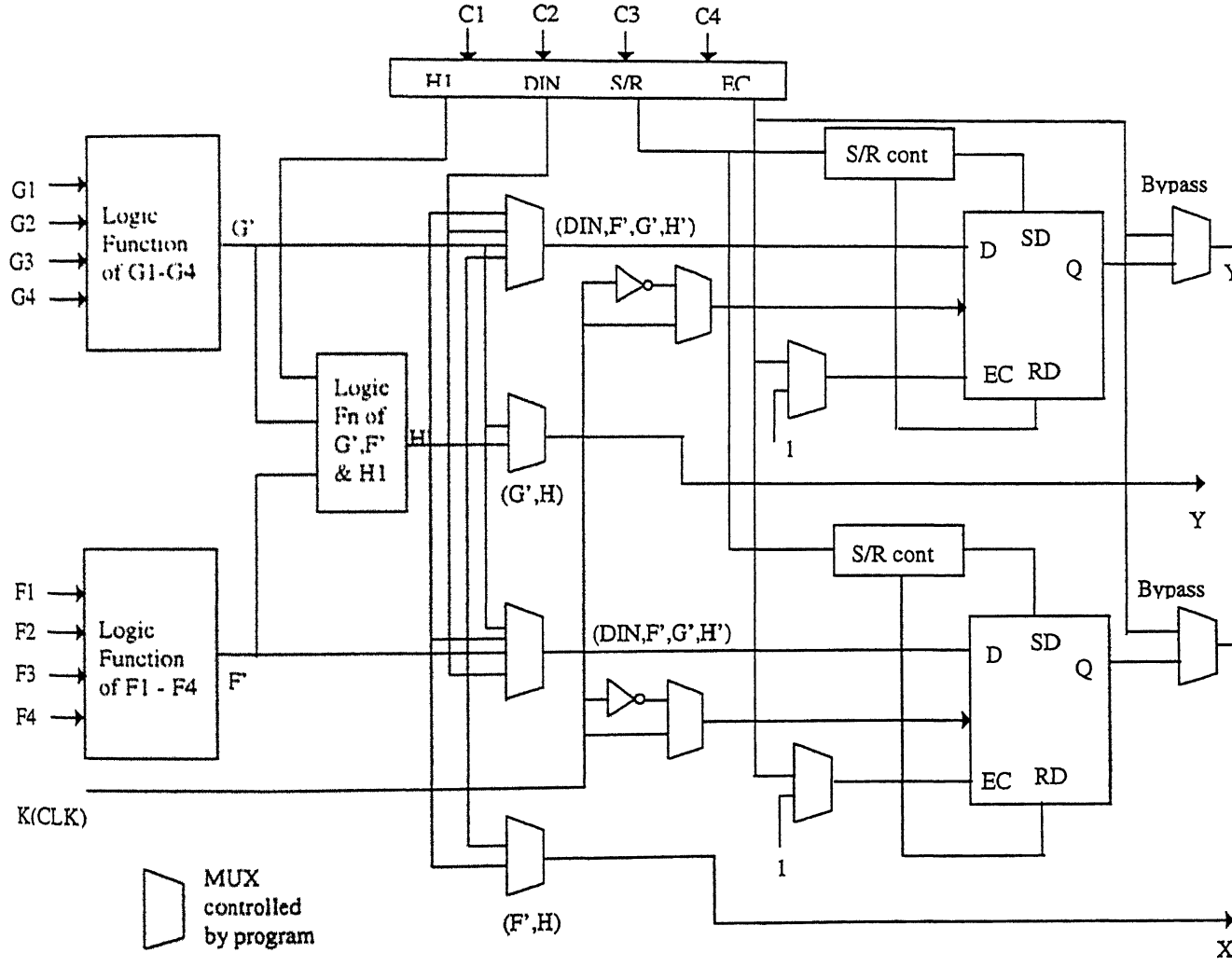


Fig 4.1 Simplified Block Diagram of CLB

4.3 Basic Modules

4.3.1 Muller_C element

The Muller_C element acts as the AND gate for the events. It is a two input and single output gate but can be generalized to any number of inputs. Functionality of Muller-C element is defined as follows when both the inputs are in the same states, then output follows the input and when the two inputs are in the different states the output retains its previous value. The synthesized schematic of the Muller-C element for two inputs is shown in Figure 4.2.

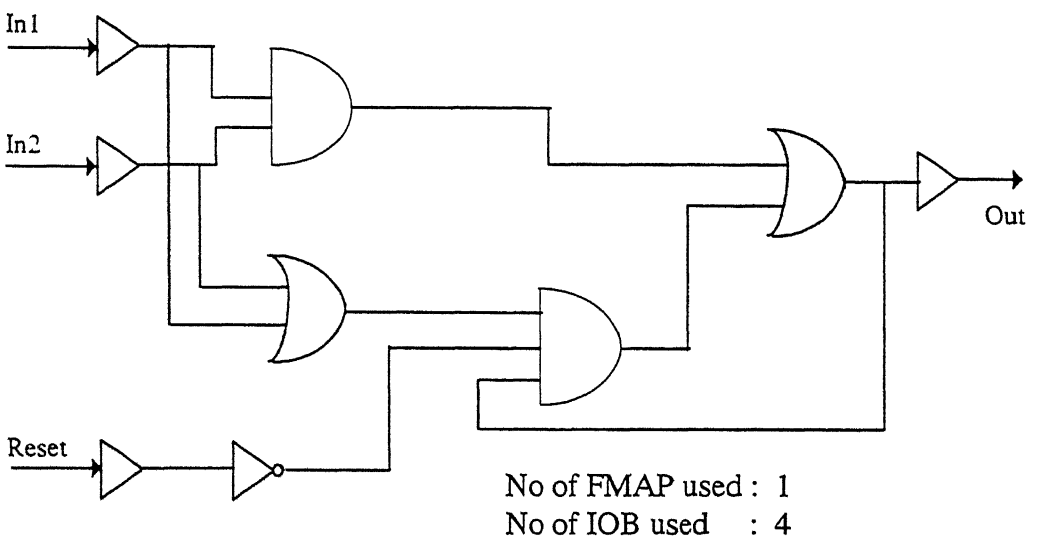


Fig 4.2 Muller C Element

4.3.2 Universal Gate

Universal gate (U-gate) accepts the inputs in the two rail format and generate an output in the single rail format. It has two inputs in the two rail formats and four outputs in the single rail format The synthesized schematic of two inputs U-gate is shown in Figure 4.3. The output transition occurs depending on the input combinations, i.e. depending on the input combinations 00, 01, 10, 11 transition appears on one of the four output line. It can realize any two input Boolean function by appropriately combining the outputs. Another notable feature of U gate is realization of functions A+B and A.B using just one U gate. However, more XOR gates are required.

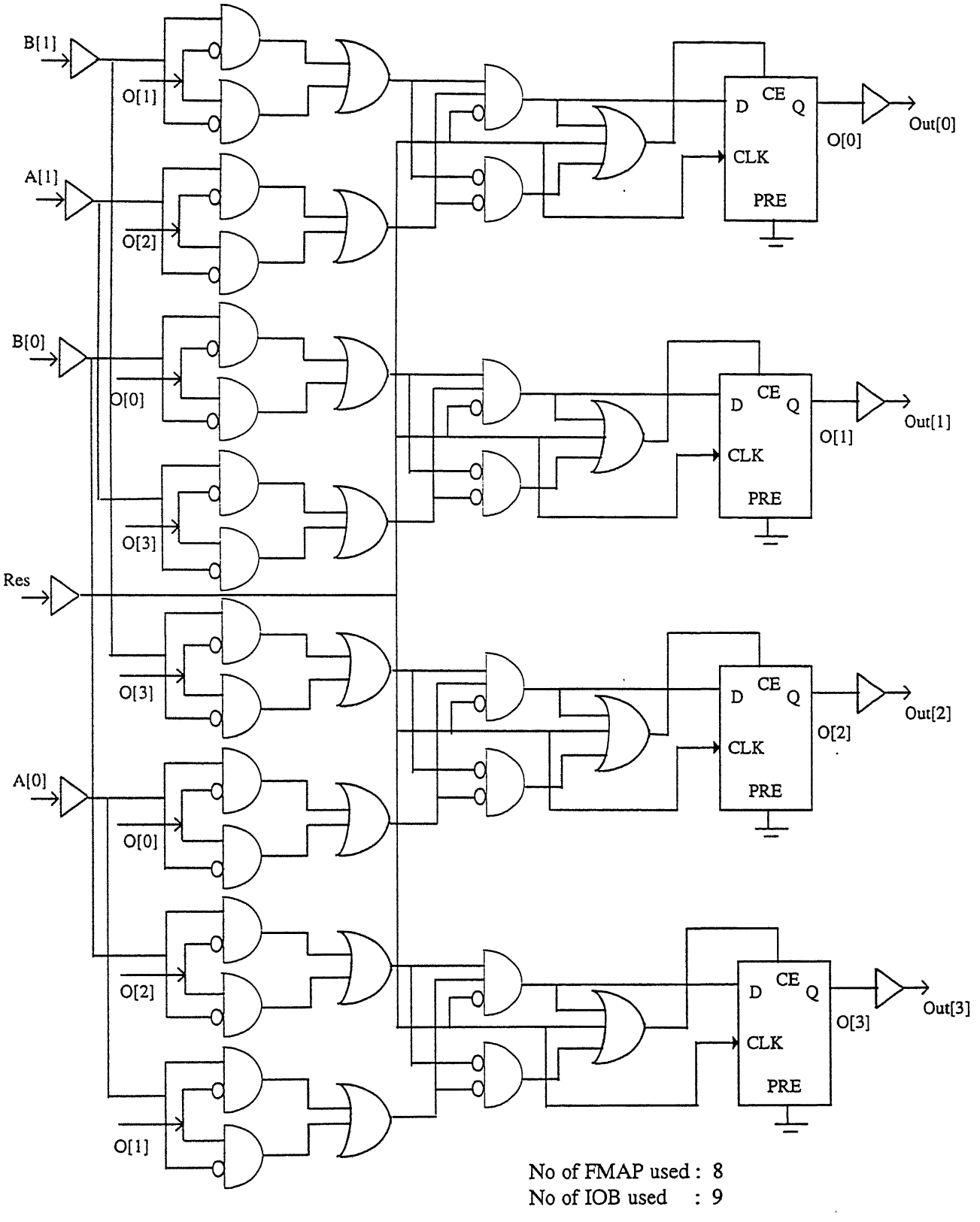
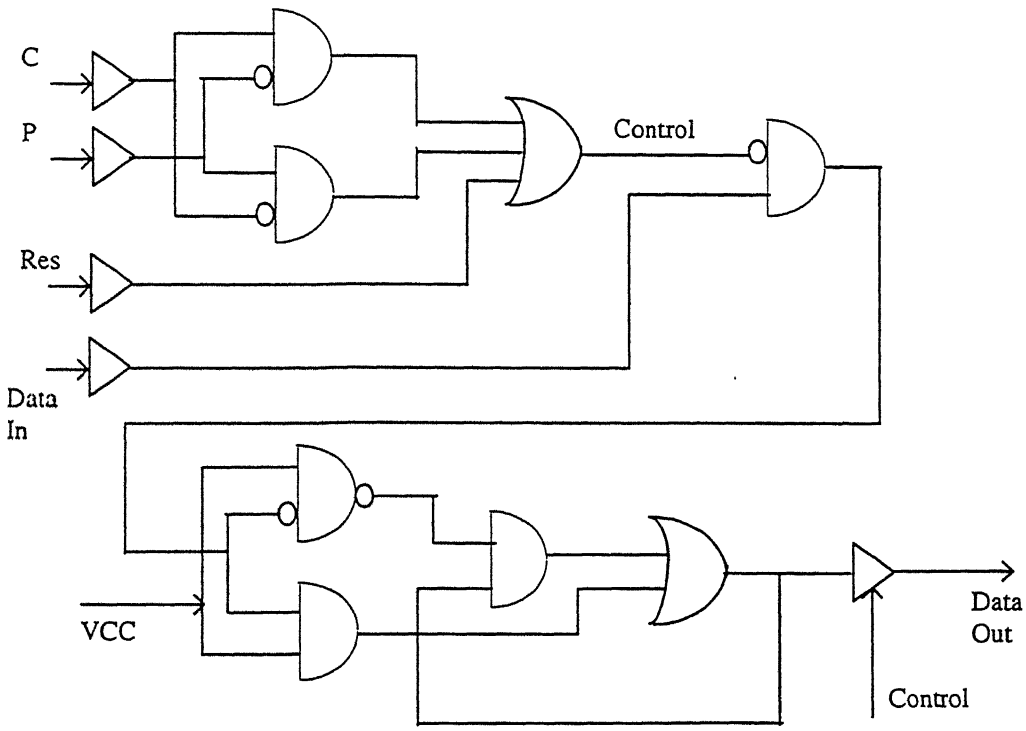


Fig 4.3 Universal Gate

4.3.3 Transition Latch

A transition latch can be used to latch an event based data. However, unlike the conventional latch in which the high and the low state of a clock signal can perform different function; it behaves identically to both rising and falling transitions. It has two control wires, ***capture*** and ***pass***, these are single rail NRZ control signal and a data line. If the ***capture*** and ***pass*** signals are in the same state, the latch is in the transparent mode. If they are different, the latch is in the capture mode. The synthesized schematic of the transition latch is as shown in Figure 4.4.



No of FMAP used : 1
No of IOB used : 5
No of TBUF : 1

Fig 4.4 Transition Latch

4.3.4 Asynchronous Register

An asynchronous register is used for storing and transfering asynchronous data in the NRZ transition signaling format. The structure of the asynchronous register is defined in [23]. It has two rail inputs; two control signals, ***data_out*** and ***data_clr;*** and two rail output ***R_out***. It also generates ***clr_ack*** signal for indicating that new input can be applied. When a transition is applied on ***data_clr*** input, old data stored in the register is cleared and a ***clr_ack*** transition is generated. Transition on the input ***data_out*** places the data in the register on the ***R_out*** terminal. The synthesized asynchronous register uses the following FPGA resources.

Number of FMAPs : 8
Number of IOBs : 8

4.3.5 Event Counter

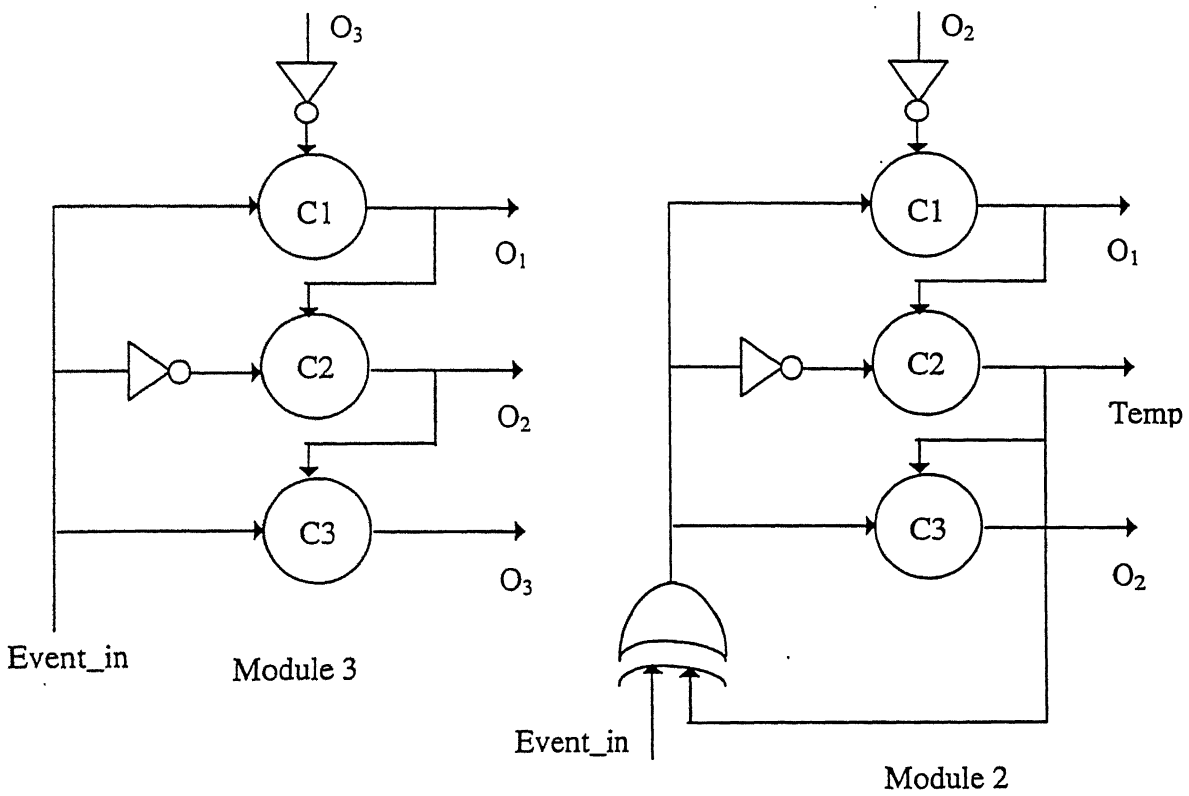
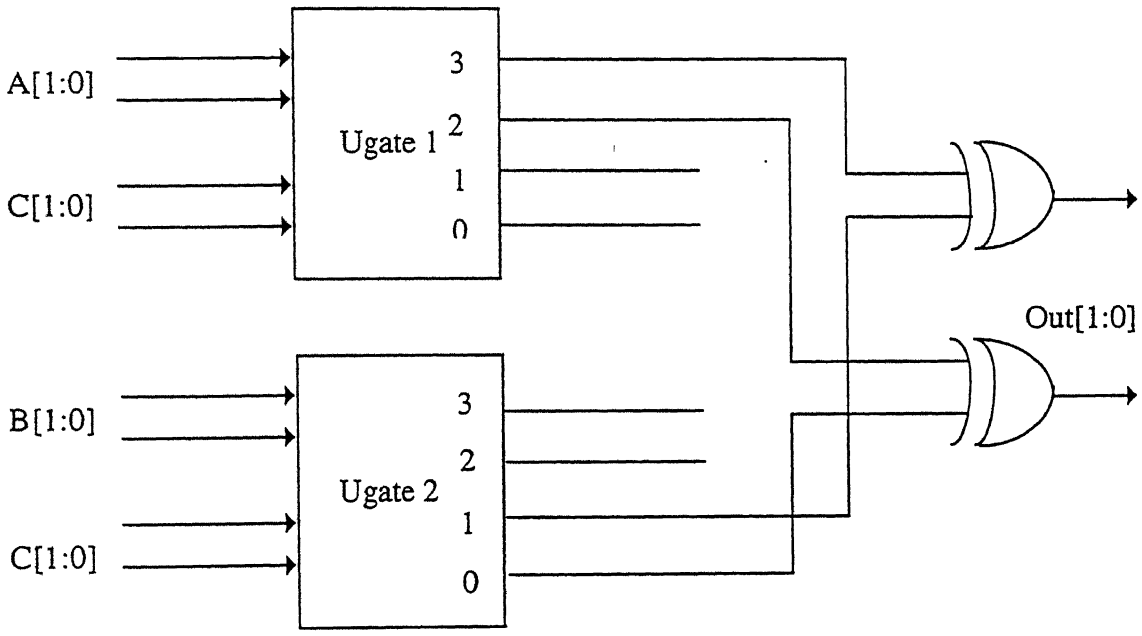


Fig 4.5 Event Counter

A module n event counter has a single one rail input, ***event_in*** and n single rail outputs O_1 to O_n. An event counter generates an output transition on one of the output lines depending on the number of transitions on the input rail, i.e., if the (i<n) transitions arrive on the input rail, a corresponding transition is generated on output O_i. The schematic of an event counter is shown in Figure 4.5. The synthesized resources are given in Table 4.1.

Module Name	No of FMAP Used	No of IOB Used
Event Counter Mod 5	5	7
Event Counter Mod 3	3	5
Event Counter Mod 2	4	4

Table 4.1 Resources Used in Event Counter



No of FMAPs : 17
No of IOBs : 9

Fig 4.6 Asynchronous Multiplexer

4.3.6 Asynchronous Multiplexer

An asynchronous multiplexer is used to multiplex the data in the transition signaling format. The schematic for a 2 input multiplexer is shown in Figure 4.6. Depending on the transition on the r_o or r_1 of the C, the transitions on data rails A or B are passed to the output respectively.

4.4 Asynchronous Circuits and Systems

4.4.1 Adder:

Asynchronous adders are used to add two numbers in the transition signaling formats. A and B are the inputs to the adder in the two rail format and C_{in} is the carry input to the full adder in the two rail format. The circuit description of the one bit adder is shown in Figure 4.7. It uses two U gates compare to 8 U gates used in [21]. We make use of the half adder approach to design this full adder, to result in a significant reduction in the number of FPGA resources.

Besides the one bit and four bit adder, other asynchronous circuits and systems which have been synthesized are the bus arbiter, circuit for implemeting point to point interconnection topology, bus decoder for fifteen outputs and bus architecture for bundled data transfer. The synthesis results of these circuits are represented in Table 4.6.

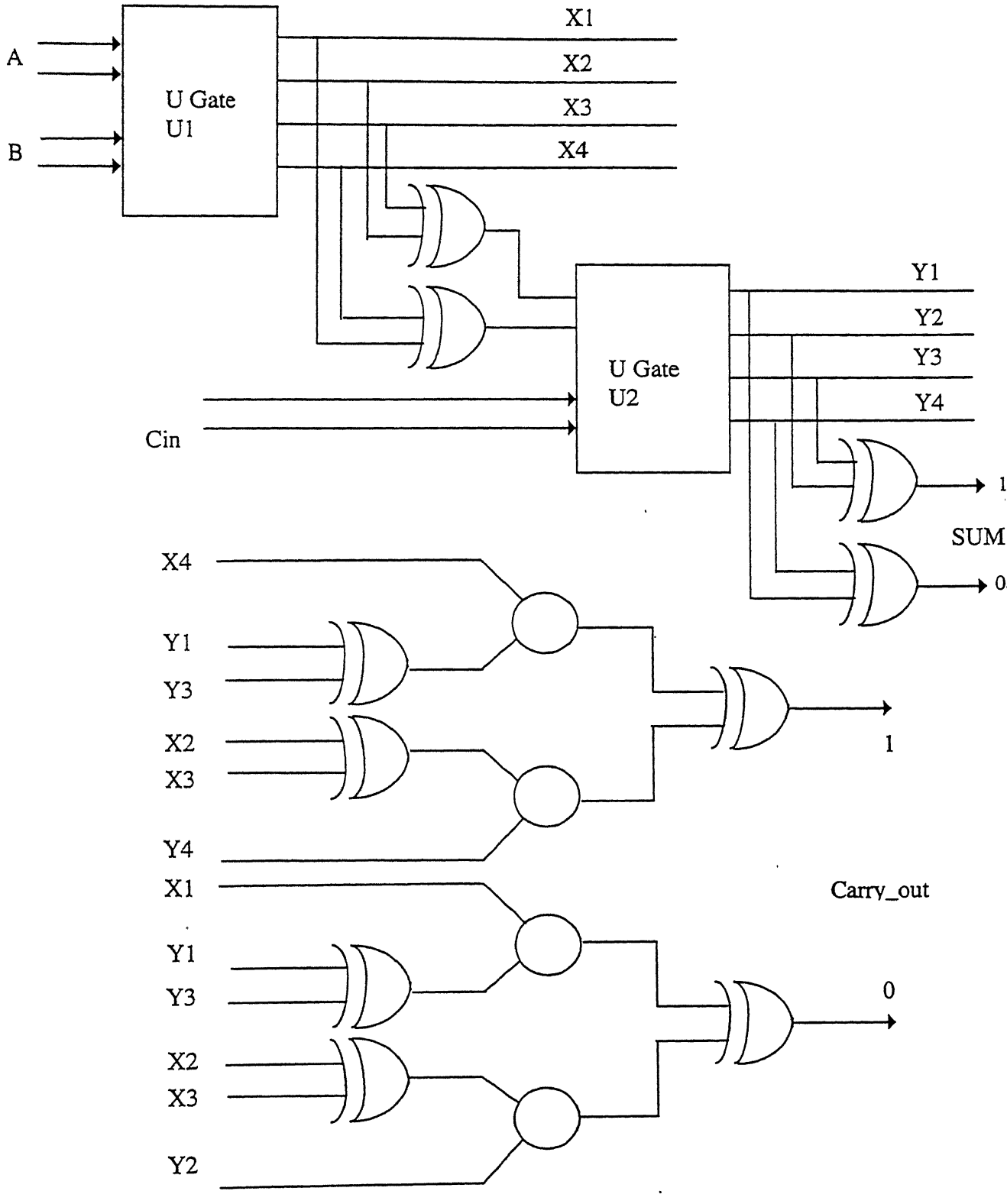


Fig 4.7 One Bit Full Adder Implementation

Module Name	No of FMAPs Used	No of IOBs Used	No of HMAPs Used	No of TBUFs Used
Adder 1 Bit	53	11	-	-
Adder 4 Bit	165	27	-	-
Decoder 15 Bit	15	17	-	-
4 bit Serial In Parallel Out Shift Register	33	16	-	-
Shifter	72	57	-	-
Bus Arbiter (2 Resources)	9	7	2	-
Point to point interconnection (2 resources)	8	9	-	-
Point to point interconnection (8 resources)	32	36	-	-
Bundled data 1 bit bus transfer (Ri <-> Rj, Ri <-> Po, Pi <-> Rj)	34	-	-	2
Bundled data 8 bit bus transfer (Ri <-> Rj, Ri <-> Po, Pi <-> Rj)	272	-	-	16

Table 4.2 Resources Used for Various Asynchronous Circuits and Systems

Chapter 5

Conclusion and Future Work

5.1 Conclusion

We have designed the asynchronous digital circuits using the two phase non return to zero transition signaling approach. This approach can be extended to design any complex asynchronous digital systems. This approach has been illustrated through the design examples; such as, asynchronous arbiter, bus architecture for bundled data transfer and adder circuits. To realise these designs we employ the basic asynchronous modules such as Muller C element, XOR gate, U gates etc., whose implemetation was presented in the previous chapter.

As the implementation of the digital systems based on asynchronous circuits gains wider acceptance , tools to carry out high level synthesis from behavioral description are becoming necessary. In this thesis we present the **Modified ELS** algorithm for synthesizing the asynchronous circuits from high level descriptions. We compare this with the existing algorithm,'**ELS** [14]. These algorithms takes care of the scheduling and allocation of the asynchronous models to realize asynchronous systems from their CDFG. These algorithms have been tested on several benchmark systems such as Differential Equation Solver, Auto Recursive Filter, Fifth Order Wave Elliptical Filter and Inverse Discrete Cosine Transform system.

Finally we have mapped the asynchronous circuits into the Xilinx 4000 series FPGA and corresponding XNF files have been generated. These XNF files can be directly

imported by the XACT backend tool and can be used for partitioning and mapping the design into an FPGA.

5.2 Future Work

Though we have presented a design methodology, to implement asynchronous digital systems, further work needs to be carried out to obtain a fully functional asynchronous system such as a microprocessor, which meets all the timing specifications.

In the bus architecture proposed in chapter 2 based on the bundled data transfer, determining the delay between a sender and a receiver through the bus is a very important design issue. This requires knowledge of the load so that the exact value of T_{PLH} and T_{PHL} can be obtained. This requires the complete layout information of the FPGA architecture, from which, by back extraction, the actual load information can be computed.

In the synthesis approach based on modified ELS algorithm, we have only considered the average delay. Further research is necessary to synthesize asynchronous circuits under stringent timing constraints. Also, most of the benchmark systems on which these algorithms have been tried are data dominant; using these algorithms on asynchronous systems which are control dominanted requires further investigation. Further, this work can be extended to include unit binding and control unit generation. The binding task assigns the operations and memory accesses to each available hardware resource. A resource, such as functional, storage or interconnection unit, can be shared by different operations, data accesses or data transfers, if they are mutually exclusive.

These algorithms can be extended to the synthesis of multi chip module (MCM) architectures, which will be the common technology by 2000. The register level design obtained from the high level synthesis of a behavioral description may be too large to fit on a single chip. In such cases, it can be partitioned into several chips to be connected on an MCM. However, in such cases system level partitioning (SLP) needs to be carried out to divide the behavioral description in the form of CDFGs into several partitions such that each partition can be synthesized onto a chip.

Appendix A

Simulations- SPICE and Verilog and Pinnauq FPGA Synthesis

A detailed transistor level simulation using SPICE has been carried out for basic asynchronous logic blocks such as Muller C element, XOR gate, transmission gate, inverter, Universal gate and Asynchronous register etc., for a 0.6 micron CMOS technolgy process, using BSIM II model parameters, obtained from ARCUS Technology Limited. All the delays for these modules have been calculated using minimum feature size (MFS) parameters and with a load of 100 fF. Some of the delays obtained are given in Table A.1

Sr No	Name of Module	Delay (ns)
1	**2 Input Muller C element**	**1.2 ns**
2	**EXOR gate**	**1.3 ns**
3	**Inverter**	**0.3 ns**
4	**Transmission Gate**	**0.3 ns**
5	**U Gate**	**2.2 ns**
6	**Aynchronous Register**	**2.3 ns**

Table A.1 Delyas Obtained for Basic Modules using SPICE

The delays obtained above have been used for simulation of the behavioral models of the above blocks in Verilog.. These delays have also been used for the

structural simulation of asynchronous arbiter, bus architecture for bundled data transfer and the adder circuits. The Verilog simulations of the above mentioned complex asynchronous structures have been carried out by first implementing the design of the basic modules and then instantiating these basic modules in the complex structures.

Finally each of the design implemented as above have been synthesized using Pinnauq, a synthesis tool, to map the register transistor level (RTL) description in Verilog into a Xilinx FPGA target architecture. All the basic modules are using the above route. The synthesized basic modules are then used to realize the complex structures. The synthesized basic modules are represented structurally in the Xilinx Neltlist Format. The structural representation of these basic modules can be instantiated in the behavioral description of complex structures, which inturn, can be synthesized.

Bibliography

[1] Scautt Hauk, " Asynchronous design methodologies : an overview ", Technical Report 93-05-07, Department of Computer Science and Engg., University of Washington, Seattle, 1993.

[2] D. W. Dobberpuhl, R. T. Witek, R. Allmon, R. Anglin, R. Bertucci, S. Britton, L. Chao, R. A. Cohrad, D. E. Devar, B. Gleseke, S. M. N. Hassoun and G. Hoeppner, "A 200mhz 64 bit dual issue CMOS microprocessor ", *IEEE Journal of Solid State Circuits*, vol. 27, pp. 1155-1167, November 1992.

[3] J. C. Ebergen, " A formal approach to desiging delay insensitive circuits ", *Distributed Computing*, Vol. 5 , no. 3, pp. 107-119, July 1991.

[4] S.Burns and A. J. Martin, "Synthesis of Self Timed Circuits by Program Transformation", Computer Science Department, California Institute of Technology, California, Private Communications

[5] S. B. Fuber, P. Day, J. D. Garside, N. C. Paver and J. V. Woods, " A micropipelined ARM", Department of Computer Science, The University of Manchester, Manchester, Private Communications.

[6] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwak and A. Takamura, " TITAC : Design of a quasi delay insensitive microprocessor ", *IEEE design & test of computers*, Vol. 11, pp. 50-63, Feberary. 1994

[7] K. Van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken and F. Schalij, " Asunchronous circuits for low power: A DCC error corrector ", *IEEE design & test of computers*, Vol. 11, pp. 22-32, Feberary 1994.

[8] I. E. Sutherland, "Micropipelines ", *Communications of the ACM*, Vol. 32 pp. 720-738, June 1989.

[9] T. H. Y. Meng, R. W. Broderson and D. G. Messershmitt, " Automative synthesis of asynchronous circuits from high level specifications ", *IEEE transaction on Computer Aided Design*, Vol. 8, pp. 1185-1205, November 1989.

[10] Cho Woo Moon, *Synthesis and verification of asynchronous circuits from graphical specifications*, PhD Thesis, Department of Electrical Engg. and Computer Science, University of California at Berkelly 1992.

[11] P. A. Beeral, C. J. Mayers and T. H. Y. Meng, " Automatic synthesis of gate level speed independent circuits", Technical Report CSE-TR-94-648, Stanford University, November 1994.

[12] S. H. Unger, *Asynchronous sequential swtiching circuits*, Wiley-Interscience, Newyork, 1969.

[13] S. B. Fuber, P. Day, J. D. Garside, N. C. Paver and J. V. Woods, " The design and evaluation of an asynchronous microprocessor", Department of Computer Science, The University of Manchester, Manchester, Private Communications.

[14] R. M. Badia and J. Cortadella, " High level synthesis of asynchronous digital circuits : scheduling strategies", UPC/DAC report no RR-92/6, Department of Computer Architecture, Polytechnic University of Catalonia, Barcelona, November 1992.

[15] R. M. Badia and J. Cortadella, " High level synthesis of asynchronous systems : scheduling and process synchronization", Department of Computer Architecture, Polytechnic University of Catalonia, Barcelona, Private Communications

[16] W. W. Plummer, " Asynchronous arbiters ", *IEEE transaction on Computers*, Vol. C-21, pp. 37-42, Janaury 1972.

[17] A. Yakovlev, A. Petrov and L. Lavagno, " A low latency asynchronous arbitration circuits ", IEEE transaction on Very Large Scale Integration (VLSI) systems, Vol. 2, No 3, pp. 372-376, September 1994.

[18] Mark B. Josephs and J. T. Yantchev, " CMOS design of the tree arbiter elemet", *IEEE transaction on Very Large Scale Integration (VLSI) systems*, Vol. 4, No 4, pp. 472-476, December 1996.

[19] C. L. Seitz, " System timing " In *Introduction to VLSI system*, C. Mead and L. Conway Eds. Readind, MA: Addison-Wesley, 1980, pp. 218-262.

[20] Win F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aparts, J. H. M. Korst, J. L. Van Meerbergen, A. V. Werf, " Improved force directed scheduling in high through-put digital signal processing ", *IEEE transaction on Computer Aided Design of Integrated Circuits and systems*, Vol. 14, pp. 945-960, Auguest 1995.

[21] K. Nanda, *Design of asynchronous digital circuits*, B.Tech Report, Department of Electrical Engg., Indian Institute of Technology Kanpur, April 1996.

[22] K.Nanda, S.K. Desai and S. K. Roy, " A new methodology for the design of asynchronous digital circuits ", in *10th International Conferrence on VLSI Design*, India 1997, pp. 342-349.

[23] S. K. Desai, *Design of asynchronous digital circuits*, M.Tech Thesis, Department of Electrical Engg., Indian Institute of Technology Kanpur, Feberary 1997.

[24] D. Gajski, W. Dutt, A.Wu and S. Lin, *High level synthesis: introduction to chip and system design*, Kluwer Acedemic Publisher, 1992.

[25] Giovanni De Micheli, *Synthesis and optimization of digital circuits*, Me-Graw Hill Newyork 1994.

[26] K. Maheswaran and Venkatesh Akella, " Hazard free implementation of self timed set for Xilinx 4000 series FPGA", Department of Electrical Engg., University of California Davis, Private Communications.

[27] K. Kucukcakar and A. C. Parker, " A methodology and design tools to support system level VLSI design ", *IEEE transaction on Very Large Scale Integration (VLSI) Systems*, Vol. 3, No 3, pp. 372-376, September 1995.

[28] D. D. Gajski and L. Ramachandran, " Introducion to high level synthesis ", *IEEE Design and test of computers*, pp. 44-53, winter 1994.

[29] *Xilinx : The programmable logic data book*, Xilinx Inc, San Jose CA, 2nd Edition, 1994.

[30] *Pinnauq : FPGA synthesis Tool Version 1.0 User's Mannual*, Silicon Automation Systems, Bangalore, 1997

[31] *Pinnauq : FPGA synthesis Tool Version 1.0 Xilinx Support Mannual*, Silicon Automation Systems, Bangalore, 1997.

[32] S.Burns, *Automatic Compilation of Cocurrent Programs into Self -Timed Circuits*, Master's Thesis, California Institute of Technology, California, 1987, Private Communications.

EE-1998-M-GAR-DES